Desarrollando con Java 8

Poker



David Pérez Cabrera

Creando una aplicación para explicar algunas de las novedades de Java 8

Como: expresiones lambda, programación funcional,

anotaciones, métodos por defecto en interfaces,

y otras curiosidades.

Versión 1.0.1

22 de Mayo de 2016

Por David Pérez Cabrera.

Este documento ha sido creado para www.javahispano.org seguramente aquí encontrarás más documentación que pueda resultar de tu interés, sí no la hay ¡se el primero en prepararla y compartirla! JavaHispano es una comunidad que se hace entre todos.

Índice

Introducción....................................4
¿Por qué el Póquer?....................................5
Licencia del documento....................................6
Licencia del código....................................7
Capítulo 0: Introducción al Póquer....................................8
Manos en el Póquer....................................8
Etapas en el Texas hold'em....................................12
Apuestas en el Texas Hold'em sin límites....................................13
El bote....................................18
Capítulo 1: Modelado básico....................................20
Capítulo 2: Algoritmos iniciales....................................36
Capítulo 3: Póquer API....................................56
Capítulo 4: Máquina de estados....................................67
Capítulo 5: Eventos y concurrencia....................................77
Capítulo 6: Temporizador....................................83
Capítulo 7: Modelo de Póquer para la máquina de estados....................................87
Capítulo 8: Estados del Póquer....................................95
Capítulo 9: Controladores para el Póquer....................................106
Capítulo 10: Creando la estrategia del jugador....................................116
Capítulo 11: La interfaz gráfica....................................119
Capítulo 12: Ensamblando componentes....................................132
Bibliografía....................................135

Introducción

El desarrollo de software es una actividad que puede llegar a ser muy complicada, requiere un gran esfuerzo y práctica. Existe una importante diversidad de paradigmas de programación, innumerables lenguajes, muchos de los cuales tienen varias especificaciones. Cada lenguaje invita a abordar los problemas de forma diferente, por muy similares que puedan llegar a ser siempre hay matices que te hacen enfocar de forma distinta el mismo problema, incluso dentro de un mismo lenguaje y especificación, cada persona construirá software de forma distinta aunque tenga el mismo fin. Si esta enorme diversidad no fuera suficiente, hay ecosistemas enteros de herramientas y tecnologías con la intención de facilitar la labor del desarrollo.

Dado el renovado interés despertado tras la especificación de Java 8 y sobre todo al gran salto dado con respecto al de especificaciones anteriores no quería desaprovechar la oportunidad de explorar las posibilidades que nos brinda.

El objetivo de este documento es desarrollar un juego de Póquer de forma didáctica utilizando Java 8 y una familia de tecnologías muy habituales en el ecosistema de Java. No es necesario tener un conocimiento profundo en estas herramientas/tecnologías ya que según vaya avanzando se van a ver algunas pinceladas para entender cómo van encajando hasta llegar al resultado final.

Cuando en 2007 apareció la Javacup, me encantó, participé con escasa fortuna, pero disfruté enormemente con la emoción de los enfrentamientos, el ambiente y la camaradería del resto de participantes. Entonces entendí como había logrado su propósito: Aprender a programar y mejorar de forma divertida con un juego. Pura genialidad, convertir el arduo proceso de aprender en un hobby. No he podido volver a participar pero me dejo una huella profunda y una espinita. Participar ahora se me hace difícil, han evolucionado mucho las tácticas de los otros participantes y soy un mal perdedor.

¿Por qué el Póquer?

Siempre he sido un gran apasionado del Ajedrez, me ha parecido un juego fascinante, enormemente complejo y divertido. Con el tiempo crecí y mi pasión por el ajedrez también lo hizo, cuando aprendí a programar descubrí a mi otra gran pasión, me han acompañado ambas desde entonces y siempre he querido tener la oportunidad de combinarlas. Esta parece una gran ocasión, sin embargo, el juego elegido es el Póquer, voy a tratar de explicar el porqué.

Antes de tener ningún contacto con el Póquer ya tenía una idea preconcebida, lo asociaba a los casinos y a las salas de juego, de modo que a priori no le veía mayor interés del que tiene lanzar unos dados y tenía muchos prejuicios por la asociación que hacía con el juego y a la parte más oscura que este tiene.

Pasó el tiempo y por casualidad di con un artículo sobre Ajedrez y Póquer, la primera reacción fue de cierto desconcierto, leyendo explicaba como los buenos ajedrecistas solían ser buenos jugadores de Póquer y esto desde mis prejuicios no era capaz de entenderlo.

No veía que relación podía tener un juego tan noble y preciso como el Ajedrez en el que juegan dos adversarios, con otro juego de azar al que ya tenía estigmatizado y en el que se enfrentan muchos jugadores. Así que me picó la curiosidad y empecé a leer un poquito sobre el Póquer, no tardé en darme cuenta de algo que ambos tenían en común, el factor psicológico y que el mejor acaba ganando cuando se disputa el número suficiente de partidas. Entonces empecé a ver el Póquer con otros ojos, no se trata sólo de azar, se trata de ganar la batalla psicológica calculando con precisión milimétrica las opciones en cada momento y además ¡a varios adversarios a la vez! Esto ya tomaba un cariz bastante más interesante. Nunca he llegado a aprender a jugar bien al Póquer, pero he pasado buenos ratos jugando con amigos.

En 1997 se marcó un hito muy importante en la historia del Ajedrez, una máquina fue capaz de ganar al campeón mundial, por un resultado muy ajustado y no sin cierta polémica, pero lo cierto es que ocurrió. Hoy tal enfrentamiento ya carece de sentido, la mejora del hardware y el perfeccionamiento que han tenido los algoritmos hacen que una máquina con el software adecuado no tenga rival (humano).

El Ajedrez desde el punto de vista computacional, para mí ha perdido interés, no sería capaz de ganar a un algoritmo sencillo con una importante base de datos de partidas, finales y un buen libro de aperturas, sin embargo, el Póquer no tiene esa problemática y debido a la gran componente psicológica que tiene, además la participación de muchos jugadores con diferentes estilos complica la forma de plantear la estrategia.

La posibilidad de hacer una estrategia de Póquer lo suficientemente buena como para jugar contra ella en una lucha equilibrada me resulta demasiado atractiva como no para elegir Póquer en lugar del Ajedrez.

Licencia del documento

El trabajo de revisión es una tarea ardua y es muy posible que se hayan escapado algunos errores, si detectas alguno por favor házmelo llegar a dperezcabrera@gmail.com para corregirlo y tratar de mejorar el documento.

Internet facilita enormemente la comunicación y difusión del conocimiento, así que gracias a que alguien ha compartido este documento, te ha podido llegar a ti, siéntete libre de compartirlo pero es importante respetar las licencias de uso, citar al autor y a la fuente.

Licencia del código

El código distribuido en este documento y en el repositorio está bajo la licencia GPL v3.0.

Se puede ver el código completo en https://github.com/dperezcabrera/jpoker

No hay software sin errores y por supuesto este no va a ser una excepción, si encuentras algún bug por favor házmelo llegar a dperezcabrera@gmail.com trataré de subsanarlo a la mayor brevedad posible.

Capítulo 0: Introducción al Póquer

El Póquer es un juego de cartas, donde los jugadores participan de forma individual realizando apuestas, las cartas que se utilizan son las de la baraja inglesa. Tiene muchas variantes, la más popular es la Texas hold'em que es la que se va a implementar en este documento. En el Texas hold'em pueden participar entre dos y diez jugadores, donde cada uno de ellos recibe dos cartas que deben mantener ocultas, en la mesa a lo largo de la ronda se van colocando cartas comunitarias boca arriba, en cada etapa se van colocando un número determinado y fijo de cartas hasta alcanzar un total de cinco. Cada jugador utiliza su pareja individual de cartas además de las comunitarias para obtener la mano más fuerte posible de entre las siete cartas, que será la que utilizará para enfrentarse al resto de jugadores.

Una mano está compuesta por cinco cartas, y en función de esas cartas se asigna una categoría, según la categoría en la que se encuentre una mano esta será más fuerte, igual o más débil que otra, cada categoría tiene un modo de desempate distinto aunque cortado por un patrón común.

Manos en el Póquer

Las posibles categorías de una mano, de más a menos fuertes son:

- Escalera de color: cinco cartas con rangos consecutivos del mismo palo.
- Póquer: cuatro cartas con el mismo rango.
- Full: Tres cartas de un rango y otras dos de otro rango.
- Color: Cinco cartas del mismo palo.
- Escalera: Cinco cartas con rangos consecutivos sin importar el palo.
- Trío: Tres cartas con el mismo rango.
- Dobles parejas: Dos cartas con un rango y otras dos con otro rango.
- Pareja: Dos cartas con el mismo rango.
- Carta más alta: Cuando la mano no tiene ninguna de las categorías anteriores.

Los palos posibles son cuatro:

Corazones (♥), Diamantes (♦), Picas (♠) y Tréboles (♣).

Los rangos de menor a mayor son:
Dos, tres, cuatro, cinco, seis, siete, ocho, nueve, diez, jack (o J), reina, rey y As.

La nomenclatura que se va a seguir para los rangos es utilizar un solo carácter: dos: 2, tres: 3, cuatro: 4, cinco: 5, seis: 6, siete: 7, ocho: 8, nueve: 9, diez: T, jack: J, reina: Q, rey: K y As: A. La nomenclatura usada para las cartas será la de utilizar dos caracteres, el primero el descrito para el rango anteriormente y el segundo el carácter especial del palo (♥, ♦, ♠, ♣).

Se da la circunstancia que el As se puede considerar como As o como 1 para componer la escalera de color o la escalera, pero cuando se considera un 1, esté será el valor de su rango y no el de As.

Veamos un ejemplo concreto usando la nomenclatura:

`A♣ K♠ Q♦ J♥ T♠` → Escalera al As, ya que el As es la carta con el rango más alto.

`5♣ 4♦ 3♦ 2♥ A♦` → Escalera al 5, ya que el 5 es la carta con el rango más alto, al considerarse el As como un 1.

El valor del rango de la carta más alta es muy importante ya que forma parte del criterio de desempate en todas las categorías. Los palos se consideran de igual valor de modo que ninguno prima sobre los demás.

Los criterios de desempate son:

- Escalera de color: El rango más alto de las cartas que la forman.
- Póquer: El rango de las cuatro cartas iguales, en caso de empate se utilizará el valor de la quinta carta.
- Full: El valor del rango de las tres cartas iguales, en caso de empate se utiliza el valor del rango de la pareja.
- Color: El valor de la carta con mayor rango y ante un empate se trataría de utilizar la siguiente para deshacerlo, hasta llegar a la quinta.
- Escalera: El rango más alto de las cartas que la forman.
- Trío: El valor del rango de las tres cartas iguales, en caso de empate se utilizará el rango de la mayor de las dos cartas restantes, y si aún así se mantiene se utilizará el rango de la última carta.
- Dobles parejas: El valor del rango de la pareja de cartas con mayor rango, en caso de empate

se utilizaría la segunda pareja con más rango y si el empate persiste ya se utilizaría la quinta carta.

- Pareja: El valor del rango de la pareja, en caso de empate, de las tres cartas restantes se utilizaran la de mayor rango, y si se mantiene el empate se pasa la siguiente, así hasta la última carta.
- Carta más alta: El valor de la carta con mayor rango y ante un empate se utiliza la siguiente y así se seguiría hasta la quinta carta si no se pudiese deshacer el empate antes.

Aunque resulte difícil, el empate es un resultado posible dentro de la comparación entre dos manos de Póquer. Veamos algunos ejemplos, tenemos los jugadores Alice, Bob, Carol y Dave en varios escenarios:

Escenario 1:
Supongamos que Alice tiene una escalera de color al cinco: [4♦ A♦ 3♦ 2♦ 5♦], una gran mano sin duda, Bob por su parte tiene una escalera de color al seis:[4♦ 6♦ 3♦ 2♦ 5♦] que es por tanto de la misma categoría que la mano de Alice, pero en el desempate, Bob ganaría, ya que su escalera es más alta. El orden final será: Bob y después Alice.

Escenario 2:
Alice tiene Póquer de cuatros y K: [4♣ K♥ 4♥ 4♠ 4♦], Bob tiene Full de 4 y reyes: [4♣ 4♥ K♠ 4♦ K♦], Carol tiene Póquer de cuatros y J: [4♣ J♥ 4♥ 4♠ 4♦], Dave tiene pareja de 3: [3♣ K♥ 3♠ 4♠ 5♦]. La categoría más alta es la de las manos de Alice y Carol, pero en el desempate Alice ganaría a Carol, ya que con los cuatros hay igualdad pero la K tiene mayor rango que J. El orden final será: Alice, Carol, Bob y finalmente Dave.

Escenario 3:
Alice tiene color: [4♦ 7♦ A♦ 8♦ J♦], Bob también tiene color: [2♦ 7♦ A♦ 8♦ J♦], Carol tiene color nuevamente: [4♣ 7♣ A♣ 8♣ J♣] y Dave tiene Escalera al Rey: [J♣ K♠ T♦ Q♥ 9♠]. La categoría más alta es Color, que es la de las manos de Alice, Bob y Carol, en el desempate hay Alice y Carol tienen As, J, ocho, siete y cuatro, por el As, J, ocho siete y dos de Bob. Con lo cual habría empate entre Alice y Carol. El orden final será: Alice y Carol, Bob y finalmente Dave.

Escenario 4:
Alice tiene la carta más alta: [9♣ K♠ 3♦ 4♥ 5♠], Bob tiene una pareja de cuatros: [4♣ K♥ 3♠ 4♠ 5♦], Carol tiene dobles parejas de K y cuatros [4♣ K♥ K♦ 4♠ J♦]. De modo que el orden final sería Carol, Bob y Alice.

En la tabla 0.1 tenemos de forma más estructurada algunas comparaciones de manos, cada fila es de la misma categoría y la columna de la izquierda tiene mayor valor que el de la derecha y las combinaciones de cada fila tienen mayor valor que las de las filas inferiores.

Mano	Categoría		Mano	Categoría
4♦ 6♦ 3♦ 2♦ 5♦	Escalera de color al 6	>	4♦ A♦ 3♦ 2♦ 5♦	Escalera de color al 5
4♣ K♥ 4♠ 4♥ 4♦	Póquer de 4 y K	>	4♣ J♥ 4♠ 4♥ 4♦	Póquer de 4 y J
4♣ 4♥ K♠ 4♦ K♦	Full de 4 y reyes	>	3♣ 3♥ A♠ 3♦ A♦	Full de 3 y ases
4♦ 7♦ A♦ 8♦ J♦	Color	>	2♦ 7♦ A♦ 8♦ J♦	Color
J♣ K♠ T♦ Q♥ A♠	Escalera al As	>	J♣ K♠ T♦ Q♥ 9♠	Escalera al Rey
4♣ 4♥ 4♦ K♠ A♦	Trío de 4	>	4♣ 4♥ 4♦ K♠ J♦	Trío de 4
4♣ K♥ K♦ 4♠ A♦	Dobles parejas de K y 4	>	4♣ K♥ K♦ 4♠ J♦	Dobles parejas de K y 4
4♣ K♥ 3♠ 4♠ 5♦	Pareja de 4	>	3♣ K♥ 3♠ 4♠ 5♦	Pareja de 3
9♣ K♠ 3♦ 4♥ 5♠	La carta más alta	>	7♣ K♠ 3♦ 4♥ 5♠	La carta más alta

Tabla 0.1

Entre los distintos jugadores hay uno que debe tener el rol de Dealer, este rol se va rotando a lo largo de las rondas. Los jugadores apuestan en un orden concreto, después de cada apuesta el siguiente en apostar es el jugador a la izquierda del anterior y el primero en apostar es el que está a la izquierda del Dealer.

Supongamos que Alice, Bob, Carol y Dave están sentados en círculo en el orden de las agujas del reloj. El Dealer es Alice, el turno de apuestas será de Bob, después Carol, luego Dave y por último Alice. Cuando termine la ronda, Bob pasará a ser el nuevo Dealer, y el turno de apuestas será de Carol, después Dave, luego Alice y finalmente Bob.

Etapas en el Texas hold'em

El juego se compone de cinco etapas: Pre-flop, Flop, Turn, River y Show down.

El juego comienza en la etapa Pre-flop, donde se reparten dos cartas a cada jugador, y se realizan dos apuestas ciegas, la del siguiente jugador al Dealer se le denomina “ciega pequeña” (Small Blind) y del jugador siguiente al anterior se le denomina “ciega grande” (Big Blind), esto es así porque son apuestas obligatorias sin saber que cartas han recibido estos jugadores. La ciega grande es el doble de la ciega pequeña, se considera un valor adecuado para la ciega grande aquella que es cincuenta veces menor que las fichas de un jugador al iniciar el juego, aunque durante una partida es normal tener incrementos del tamaño de las ciegas, con el objetivo de agilizarla y que esta no se eternice.

Hay un caso especial, y ocurre cuando sólo hay dos jugadores, entonces el Dealer debería apostar la ciega grande, pero lo que ocurre es que no hay ciega pequeña y sólo hay una apuesta ciega, la ciega grande del jugador que no es el Dealer. Como hay jugadores que apuestan sin conocer sus cartas, el primero en iniciar la ronda de apuestas será siguiente que no haya tenido que apostar una ciega, en caso de ser sólo dos jugadores, el turno será del Dealer.

En la etapa Flop se colocan sobre la mesa tres cartas boca arriba, serán las tres primeras cartas comunitarias y se inicia la ronda de apuestas comenzando por el jugador a la izquierda del Dealer que siga participando en el juego.

En la etapa Turn se coloca sobre la mesa otra carta boca arriba, es la cuarta carta comunitaria y siguiendo el mismo criterio que en la etapa Flop se comienza la ronda de apuestas.

En la etapa River se coloca la última carta comunitaria sobre la mesa y se vuelve a iniciar la ronda de apuestas con el mismo criterio que en las dos etapas anteriores.

En la última etapa, la de la hora de mostrar las cartas para evaluar quién tiene la mejor mano, se le llama Show down.

La única etapa que en todas las rondas se mantiene es la primera, para ir pasando a la siguiente es necesario que queden dos o más jugadores activos en la partida. Cualquier jugador activo puede pasar en una ronda de apuestas cuando llegue su turno, a partir de ahí quedará a la espera de que termine la ronda para volver a participar en la siguiente.

Apuestas en el Texas Hold'em sin límites

Ya hemos hablado de las apuestas ciegas obligatorias, ahora toca el turno a las voluntarias en la modalidad de Texas hold'em sin límites. Se denomina sin límites porque un jugador puede apostar todo lo que desee siempre y cuando supere la apuesta mínima o apueste todas las fichas que le queden, se trata de “ir con todo” o All-in. La apuesta mínima a igualar será la más alta de las que existan previamente, al iniciar la ronda será la ciega grande y el turno corresponderá al jugador que esté a la izquierda del que haya tenido que realizar la ciega grande, este jugador tiene varias opciones, como por ejemplo pasar, igualar o subir.

Si ha decidido pasar, el jugador dejará de estar activo hasta la siguiente ronda. Pero tiene una oportunidad de ganar algo siempre y cuando haya igualado o superado a algún jugador que lo haya apostado todo. En cuyo caso se deberá llegar forzosamente a la última etapa, ver las cartas y determinar que jugadores ganan y cuánto.

Si ha decidido igualar, quedará a la espera de ver que han hecho el resto de jugadores, si ninguno ha subido, se da por terminada la ronda de apuestas y se pasa a la siguiente etapa; si por el contrario alguien ha decidido subir, vuelve al punto de partida, es decir, o pasa y abandona la ronda, iguala o sube. Este ciclo se repetirá hasta que todos los jugadores activos y que no estén en estado All-in hayan apostado exactamente la misma cantidad.

Si ha decidido subir, también quedará a la espera de ver que han hecho el resto de jugadores, si ninguno más lo ha igualado y no hay nadie en All-in, habrá ganado la ronda y se quedará con el bote, si nadie ha igualado la apuesta pero hay jugadores en All-in, se levantaran todas las cartas comunitarias y se llegará a la etapa de la hora de la verdad (ShowDown) para determinar quién gana y como se reparte el bote. Si el resto de jugadores han igualado su apuesta y ninguno la ha subido, se pasará a la siguiente etapa, y si alguien la ha subido, volvemos al estado anterior donde hay que determinar si se pasa, iguala o se vuelve a subir la apuesta.

Cuando alguien ha apostado todo, puede ocurrir lo siguiente, que nadie más lo haya hecho ni haya igualado su apuesta y hayan pasado, con lo que este jugador ganaría el bote. Qué un jugador lo haya igualado, entonces se mostrarán las cartas comunitarias y las de estos jugadores para determinar quién gana y cuánto. Que alguien apueste más, entonces el jugador inicial solo optará al bote de las fichas que le hayan igualado o se hayan retirado.

Con la diversidad de casuísticas que hay, cada jugador puede optar a los botes contengan una apuesta All-in y que sean inferiores o iguales a su apuesta máxima, y si no los hay será un único bote y el ganador será el único jugador que no se haya retirado y si hay más de uno, entonces tendrán que mostrar sus cartas para ver quién es el ganador.

El jugador que al inicio de la ronda hizo una apuesta ciega tiene derecho a apostar cuando el resto de los jugadores lo hayan hecho, si igualaron su apuesta, él puede mantenerse y pasar a la etapa Flop, o subirla y continuar con la ronda de apuestas.

Veamos un ejemplo: Es la primera ronda, ya se ha repartido las cartas individuales y se han hecho las apuestas obligatorias.

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer	2º	Alice	1000	1000		Esperando su turno
Small Blind	3º	Bob	1000	990	[10] ciega	Esperando su turno
Big Blind	4º	Carol	500	480	[20] ciega	Esperando su turno
	1º	Dave	100	100		Turno de apuesta

Es el turno de Dave, para continuar debería apostar 20 fichas, si se retira, el juego continuará con el resto de jugadores, si decide igualar, debe apostar las 20 fichas para igualar la apuesta más alta (la ciega grande de Carol). Si desease apostar más de 20 fichas, estaría elevando la apuesta más alta, si apostase las 100 fichas que tiene estaría elevando la apuesta y jugando con todo (All-in).

Continuaremos con el ejemplo, Dave ha decidido igualar la apuesta:

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer	1º	Alice	1000	1000		Turno de apuesta
Small Blind	2º	Bob	1000	990	[10] ciega	Esperando su turno
Big Blind	3º	Carol	500	480	[20] ciega	Esperando su turno
		Dave	100	80	20	Check

Ahora Alice tiene el turno y puede pasar (sin apostar nada), igualar (apostando 20), o subir la apuesta (apostando de 21-1000). Dave no volverá a apostar en esta ronda si ningún jugador eleva la apuesta. Alice decide apostar 30 y elevar la apuesta.

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer		Alice	1000	970	30	Raise
Small Blind	1º	Bob	1000	990	[10] ciega	Turno de apuesta
Big Blind	2º	Carol	500	480	[20] ciega	Esperando su turno
	3º	Dave	100	80	20	Check

El turno es de Bob, y este puede igualar apostando 20, ya que tiene una apuesta previa de 10, puede pasar o puede subir, en este punto, Bob ha decidido que es mejor pasar.

Vemos como queda la situación tras el turno de Bob:

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer		Alice	1000	970	30	Raise
Small Blind		Bob	1000	990	[10] ciega	Fold
Big Blind	1º	Carol	500	480	[20] ciega	Turno de apuesta
	2º	Dave	100	80	20	Check

Ahora le toca el turno a Carol, tiene la opción de pasar, igualar apostando 10, o subir la apuesta apostando más de 10. Carol quiere seguir jugando pero prefiere no subir la apuesta de modo que apuesta 10:

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer		Alice	1000	970	30	Raise
Small Blind		Bob	1000	990	[10] ciega	Fold
Big Blind		Carol	500	470	30	Call
	1º	Dave	100	80	20	Turno de apuesta

Dave es quién tiene el turno ahora, le quedan pocas fichas y tiene buenas cartas y prefiere arriesgar lo que le queda, de modo que apuesta todo. Como ha subido la apuesta, los jugadores que siguen activos (Alice y Carol) tendrán que seguir jugando en la actual ronda de apuestas.

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
Dealer	1º	Alice	1000	970	30	Turno de apuesta
Small Blind		Bob	1000	990	[10] ciega	Fold
Big Blind	2º	Carol	500	470	30	Call
		Dave	100	0	100	All-in

Alice cree que con sus cartas tiene pocas posibilidades de tener una mano fuerte, como Dave ha apostado todo, si lo iguala deberá llegar al momento de la verdad y mostrar sus cartas, por lo que decide pasar.

Tras su turno, le toca a Carol:

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
	1º	Alice	1000	970		Fold
Dealer	2º	Bob	1000	990	[10] ciega	Fold
Small Blind	3º	Carol	500	470	30	Call
Big Blind	4º	Dave	100	0	[20] ciega	All-in

Carol puede pasar, con lo que Dave ganaría la ronda, o igualar e ir directamente a la hora de la verdad y ver las cartas ya que no hay ningún otro jugador activo. Como decide pasar, Dave gana la ronda.

El estado en la siguiente ronda queda así:

Estado partida			Pre-flop		Bote:	0
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
	1º	Alice	970	970		Turno de apuesta
Dealer	2º	Bob	990	990		Esperando su turno
Small Blind	3º	Carol	470	460	[10] ciega	Esperando su turno
Big Blind	4º	Dave	170	150	[20] ciega	All-in

Es el turno de Alice, que decide igualar apostando 20, Bob por su parte pasa, Carol iguala subiendo 10 y Dave que no necesita subir la apuesta para continuar, la mantiene.

Se colocan las tres primeras cartas comunitarias y se pasa a la etapa de Flop, en el bote hay 60 fichas y Bob ya está fuera de la ronda. Como en esta etapa no hay apuestas obligatorias, es el turno del siguiente jugador tras el Dealer: Carol, después le tocará el turno a Dave y finalmente a Alice.

Estado partida			Flop		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
	3º	Alice	970	950		Check
Dealer		Bob	990	990		Fold
Small Blind	1º	Carol	470	450		Check
Big Blind	2º	Dave	170	150		Check

Carol decide mantener su apuesta, Dave también y Alice hace lo mismo, con lo cual se agrega una nueva carta comunitaria.

Estado partida			Turn		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
	3º	Alice	970	950		Check
Dealer		Bob	990	990		Fold
Small Blind	1º	Carol	470	450		Check
Big Blind	2º	Dave	170	150		Check

Pasamos a la siguiente etapa y nuevamente se repiten los turnos de la etapa anterior.

Estado partida			River		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
	3º	Alice	970	950		Check
Dealer		Bob	990	990		Fold
Small Blind	1º	Carol	470	450		Check
Big Blind	2º	Dave	170	150		Check

Carol y Dave deciden mantener su apuesta, sin embargo Alice sube su apuesta en 450 para forzar a sus rivales a abandonar o a jugárselo todo.

Estado partida			Pre-flop		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
		Alice	970	500	450	Raise
Dealer		Bob	990	990		Fold
Small Blind	1º	Carol	470	450		Check
Big Blind	2º	Dave	170	150		Check

Carol acepta y apuesta todo, Dave por su lado decide hacer lo mismo y apuesta sus 150.

Estado partida			Pre-flop		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
		Alice	970	500	450	Raise
Dealer		Bob	990	990		Fold
Small Blind		Carol	470	0	450	All-in
Big Blind		Dave	170	0	450	All-in

El desenlace lo veremos en el siguiente apartado.

El bote

Como hemos podido ver, puede existir más de un bote, como mínimo habrá uno y luego pueden aparecer botes adicionales por cada jugador que vaya con todo (siempre que sean cuantías diferentes). Los jugadores optarán a todos los botes que su apuesta les haya permitido cubrir, veamos como concluye el ejemplo del apartado anterior:

Estado partida			Pre-flop		Bote:	60
	Turno	Jugador	Fichas iniciales	Fichas	Apuesta	Estado
		Alice	970	500	450	Raise
Dealer		Bob	990	990		Fold
Small Blind		Carol	470	0	450	All-in
Big Blind		Dave	170	0	450	All-in

Aquí habrá dos botes, el primero de 510 fichas por el que compiten Alice, Carol y Dave, (60 de la ronda previa + 150 de cada uno de los tres). El segundo bote de 600 fichas, se lo jugarán entre Alice y Carol, (300 de Alice y 300 de Carol, las que sobran de igualar el bote anterior).

Ahora toca ver como se reparten los botes:

Suposición 1:
Alice tiene la mejor mano de los tres, con lo cual se queda con todos los botes.

Suposición 2:
Carol tiene la mejor mano de los tres, con lo cual se queda con todos los botes, al igual que Alice en la suposición anterior.

Suposición 3:
Dave tiene la mejor mano de los tres, con lo cual se queda con el primer bote. El segundo bote dependerá de las Alice y Carol.

Suposición 3.1: Alice tiene mejor mano que Carol, con lo cual se queda con el segundo bote.
Suposición 3.2: Carol tiene mejor mano que Alice con lo cual se queda con el segundo bote.
Suposición 3.3: Alice y Carol tienen manos iguales, así que se reparten el segundo bote.

Suposición 4:
Alice y Carol tienen las mejores manos y son iguales, con lo cual se reparten ambos botes.

Suposición 5:
Alice y Dave tienen la mejor mano y son iguales, con lo cual se reparten el primer bote y Alice se queda con el segundo.

Suposición 6:
Carol y Dave tienen la mejor mano y son iguales, con lo cual se reparten el primer bote y Carol se queda con el segundo.

Veamos otros ejemplos, con otras rondas de apuestas.

Escenario 1:
Alice ha apostado 100, Bob otros 100, Carol ha pasado y Dave ha apostado 50 pero no ha querido igualar a 100 por lo que ha acabado pasando, el resultado es que sólo hay un bote con 250 fichas, que se jugarán entre Alice y Bob.

Escenario 2:
Alice ha apostado 100, Bob otros 100, Carol ha apostado 10 y después no ha querido igualar y Dave ha apostado 50, que es todo lo que tiene. Hay dos botes, el primero se decidirá entre Alice, Bob y Dave y cuenta con 160 fichas (50 de Alice, 50 de Bob, 10 de Carol y 50 de Dave) y no se podrá incrementar. El segundo bote tiene 100 fichas (50 de Alice y 50 de Bob), podrá seguir incrementándose y se lo jugarán entre Alice y Bob, de tal forma que si uno pasa el otro se queda con el segundo bote.

Escenario 3:
Alice ha apostado 100, Bob otros 80 y luego pasa, Carol ha apostado 70 que es todo lo que tiene y Dave ha apostado sus 50 últimas fichas. Habría tres botes, el primero con 200 (50 de cada uno), donde se lo jugarían entre todos, el segundo bote con 60 (20 de Alice, 20 de Bob y 20 de Carol), que se lo jugarían entre ellos tres, y un tercer bote con 40 (30 de Alice y 10 de Bob), que será para Alice ya que Bob se ha retirado. Con lo cual, Alice participa en todos y ha ganado el tercero, Bob optará a ganar los dos primeros botes al igual que Carol y Dave sólo podrá ganar el primer bote.

Aquí concluye el capítulo de introducción al Póquer en su variante Texas hold'em. Hay mucha literatura al respecto, más extensa y completa que aborda cuestiones que no hemos tratado, como estrategia, tipos de jugadores, metodología de apuestas, etc. pero el objetivo de este documento no es explicar el juego con todo lujo de detalle, no obstante para comprender el funcionamiento del software hay que conocer la lógica que pretende implementar ya que percatarse del trasfondo del diseño requiere entender al menos los conceptos básicos. Este es el motivo de la inclusión de este capítulo 0.

Capítulo 1: Modelado básico

En primer lugar vamos a modelar el componente más elemental del juego: las Cartas. Siguiendo el ejemplo propuesto por Joshua Bloch en su libro Effective Java, definiremos la clase "*Card*" y los enumerados que representen el rango y el palo de una carta como enumerados anidados en la clase inicial. También vamos a definir los atributos que debe tener una carta: un atributo de tipo palo y otro atributo de tipo rango.

Otras consideraciones a tener en cuenta son: Definición de la clase *Card* como no extensible (utilizando el modificador *final*) e inmutable definiendo sus atributos (inmutables también) con el mismo modificador *final*. De este modo evitamos comportamientos indeseados derivados por una clase hija y tiene una estructura fija al ser inmutable.

Agregando los getters de los atributos quedará el código del ejemplo 1.1.

```
public final class Card {

    public enum Suit {

        SPADE, HEART, DIAMOND, CLUB
    }

    public enum Rank {

        TWO, TRHEE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE
    }

    private final Suit suit;
    private final Rank rank;

    public Card(Suit suit, Rank rank) {
        this.suit = suit;
        this.rank = rank;
    }

    public Suit getSuit() {
        return suit;
    }

    public Rank getRank() {
        return rank;
    }
}
```

Ejemplo 1.1

En la definición de los enumerados de tipo *Rank*, estos están convenientemente ordenados para proporcionar un modo sencillo de comparación utilizando el método *ordinal()*. Este método devuelve un número entero referido a la posición del elemento respecto al resto del conjunto de enumerados comenzando por '0'. Por poner un ejemplo:

Rank.TWO.ordinal() → 0,
Rank.FOUR.ordinal() → 2,
Rank.JACK.ordinal() → 9,
Rank.ACE.ordinal() → 12.

Otro concepto importante a definir es una mano, una mano es una combinación de cinco cartas distintas, se clasifican por categorías y dentro de cada categoría se clasifican por las cartas individuales que la componen. Esto permite comparar distintas manos para determinar cuál es la ganadora o ganadoras si hubiera más de una.

Vamos a definir una clase utilidad llamada "*Hands*" que contenga una constante llamada *CARDS* para indicar el número de cartas que componen una mano y un tipo enumerado anidado que represente el tipo de categoría de una mano, en un alarde de creatividad lo llamaremos "*Type*".

Las clases de utilidades no deben poder ser instanciadas, por este motivo se crea un único constructor privado y se definen como finales para que tampoco puedan ser extendidas, de modo que sólo contienen métodos estáticos, constantes y clases o enumerados anidados.

Siguiendo el mismo criterio que con *Card.Rank* en el ejemplo 1.1, se ha realizado una definición con los tipos de categoría ordenados de menor a mayor valor.

La clase "*Hands*" queda reflejada en el ejemplo 1.2.

```
public final class Hands {
    public static final int CARDS = 5;

    public enum Type {

        HIGH_CARD,
        ONE_PAIR,
        TWO_PAIR,
        THREE_OF_A_KIND,
        STRAIGHT,
        FLUSH,
        FULL_HOUSE,
        FOUR_OF_A_KIND,
        STRAIGHT_FLUSH
    }

    private Hands() {
    }
}
```

Ejemplo 1.2

La clase *Card* ha quedado muy compacta pero parece que le falta algo, si queremos escribir una funcionalidad y queremos ver que valores de las instancias de *Card* en la salida estándar del ejemplo 1.3.

```
public class Main {
    public static void main(String[] args) {
        Card card = new Card(Card.Suit.CLUB, Card.Rank.ACE);
        System.out.println("As de tréboles: " + card);
    }
}
```

```
As de tréboles: org.poker.api.core.Card@33
```

Ejemplo 1.3

No parece que el texto de la ejecución 1.3 org.poker.api.core.Card@33 sea muy descriptivo de lo que pretendemos presentar, este texto viene de la implementación de la clase *Object*, se muestra la paquetería completa de la clase y un identificador de la referencia del objeto precedido por la @.

Debemos sobrescribir el método *toString*() de la clase *Card* para poder obtener un resultado más descriptivo que nos pueda resultar útil.



Utilizar la salida estándar para mensajes de depuración es una práctica desaconsejada. Se recomienda utilizar un logger para este fin, ya que de este modo se puede configurar donde se quiere que vaya la salida y se puede activar/desactivar por configuración sin necesidad de modificar el código.

Un logger es una clase que registra mensajes, tiene varios niveles de prioridad y permite elegir la salida y el nivel de prioridad de los mensajes que se muestran por configuración. Los niveles de prioridad dependen del framework escogido. Java posee un logger estándar (*java.util.logging.Logger)* definido en la JSR-47[1], pero personalmente prefiero otras alternativas. Las más populares son log4j, slf4j , logback, log4j 2 o commons logging.

Reescribamos el ejemplo anterior atendiendo a la recomendación, para ello utilizaremos un logger, aunque log4j es el más popular, tengo preferencia por **Simple Log Facade for Java** (**slf4j**), junto a una implementación del mismo **logback**. En concreto vamos a utilizar la versión 1.7.7 de slf4j y la 1.1.2 de logback. No voy a realizar ningún tipo de configuración de este componente, con lo cual vamos a ver la salida que tiene por defecto.

1 Una JSR (Java Specification Request) es una especificación formal de un tecnología propuesta para ser añadidas a la plataforma Java. https://jcp.org/en/home/index, Ver Java Community Process (JCP).

En el ejemplo 1.4 podemos ver como ha quedado esta implementación:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Main {

    private static final Logger LOGGER = LoggerFactory.getLogger(Main.class);

    public static void main(String[] args) {
        Card card = new Card(Card.Suit.CLUB, Card.Rank.ACE);
        LOGGER.info("As de tréboles: {}", card);
    }
}
```

```
22:00:00.000 [main] INFO org.poker.api.core.Main - As de tréboles:
org.poker.api.core.Card@33
```

Ejemplo 1.4

El primer cambio es la inclusión de las librerías de la paquetería de slf4j, la definición de un objeto constante de tipo logger y de la sustitución de *System.out.println* por la llamada al método info del objeto logger, y por la forma de pasar argumentos al texto del logger.

También ocurre un cambio en la salida, aparece información adicional precediendo a la salida que pretendíamos mostrar, esta información es relativa al contexto de ejecución de la llamada al logger, por supuesto es configurable y tras el delimitador “ - “ se muestra el texto de salida.

De momento no voy a ahondar más los conceptos que envuelven al logger, este es un tema que da para escribir varios libros, únicamente añadir la convención para nombrar los logger; deben ser privados, estáticos, finales e inicializarse con la clase en la que van a utilizarse. Indicar que se suelen definir al principio de la clase.

Volviendo a la cuestión, debemos sobrescribir el método *toString*() de la clase *Card* para conseguir un resultado más descriptivo. Podemos utilizar una utilidad del IDE que estemos utilizando, en este caso Netbeans.

El resultado lo tenemos en el ejemplo 1.5.

```
    @Override
    public String toString() {
        return "Card{" + "suit=" + suit + ", rank=" + rank + '}';
    }
```

Ejemplo 1.5

El método *toString()* viene precedido por una anotación *@Override* para indicar que está sobrescrito.

Si volvemos a ejecutar el ejemplo 1.4 tenemos el resultado en la ejecución 1.1.

```
22:00:00.000 [main] INFO  org.poker.api.core.Main - As de treboles: Card{suit=CLUB, rank=ACE}
```

Ejecución 1.1

Ya tenemos un mensaje legible, podemos observar cierto texto que no hemos indicado nosotros como “CLUB” y “ACE”, esto se debe a la implementación *toString*() del tipo enumerado.

El resultado es descriptivo, quizás un poco extenso si pretendemos mostrar en una línea un conjunto de cartas, por este motivo vamos a reescribir este método para obtener un resultado más corto como: “*AC*” para “*Card{suit=CLUB}, rank=ACE}*”. Como primer carácter vamos a utilizar un número para los Rangos del 2 al 9, y para el resto podemos utilizar la primera letra del enumerado. Como segundo carácter vamos a utilizar el primer carácter del enumerado *Suit*.

Por simplificar vamos a definir una constante donde cada posición se corresponda con el carácter adecuado en el caso del rango. El resultado podemos verlo en el ejemplo 1.6.

```
    private static final String STRING_RANK_CARDS = "23456789TJQKA";

    @Override
    public String toString() {
        int rankValue = rank.ordinal();
        return STRING_RANK_CARDS.substring(rankValue, rankValue + 1)
                .concat(suit.name().substring(0, 1));
    }
```

Ejemplo 1.6

Una nueva ejecución del ejemplo 1.2 nos devuelve:

```
22:00:00.000 [main] INFO  org.poker.api.core.Main - As de treboles: AC
```

Ejecución 1.2

Ya tenemos el resultado esperado, ahora vamos a utilizar la clase *Card* en un nuevo ejemplo, vamos a crear un conjunto de cartas y vamos a insertar una nueva carta si esta no estaba en el conjunto.

El ejemplo 1.7 ilustra el código empleado para el uso descrito anteriormente, utiliza la interfaz *Set* y la clase *HashSet* como implementación, ambas son de Java estándar. Para definir el tipo de colección se utilizan los genéricos de Java, se definen el tipo con *Set<Card>* y para *HashSet<>* se utiliza el operador Diamante que permite averiguar el tipo en compilación.

Hay otras librerías de colecciones no estándar que extienden la funcionalidad de las colecciones estándar como commons collections de Apache o guava de Google.

```java
import java.util.HashSet;
import java.util.Set;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Main {
    private static final Logger LOGGER = LoggerFactory.getLogger(Main.class);

    private static void insert(Set<Card> cards, Card card){
        if (!cards.contains(card)){
            LOGGER.debug("insertamos la carta: {}", card);
            cards.add(card);
        } else {
            LOGGER.debug("la carta: {} ya estaba en el conjunto", card);
        }
    }

    public static void main(String[] args) {
        Set<Card> cards = new HashSet<>();
        Card[] cards2Insert = {
            new Card(Card.Suit.CLUB, Card.Rank.ACE),
            new Card(Card.Suit.CLUB, Card.Rank.TWO),
            new Card(Card.Suit.CLUB, Card.Rank.TRHEE),
            new Card(Card.Suit.CLUB, Card.Rank.ACE),
        };
        for (Card card : cards2Insert) {
            insert(cards, card);
        }
    }
}
```

```
22:00:00.000 [main] DEBUG  org.poker.api.core.Main - insertamos la carta: AC
22:00:00.001 [main] DEBUG  org.poker.api.core.Main - insertamos la carta: 2C
22:00:00.002 [main] DEBUG  org.poker.api.core.Main - insertamos la carta: 3C
22:00:00.003 [main] DEBUG  org.poker.api.core.Main - insertamos la carta: AC
```

Ejemplo 1.7

El resultado final no se corresponde con el esperado, El As de tréboles se inserta dos veces a pesar de que el método *insert()* comprueba si ya está en el conjunto para no volverlo a insertar, esto se debe que para comparar cartas se utilizan los métodos heredados *equals()* y *hashCode()* de la clase *Object*.

Veamos como sobrescribir estos métodos en la clase *Card* en el ejemplo 1.8 para conseguir que el algoritmo del ejemplo 1.7 tenga el comportamiento deseado.

```
@Override
public int hashCode() {
    return rank.ordinal() * Suit.values().length + suit.ordinal();
}

@Override
public boolean equals(Object obj) {
    boolean result = true;
    if (this != obj) {
        result = false;
        if (obj != null && getClass() == obj.getClass()) {
            result = hashCode() == ((Card) obj).hashCode();
        }
    }
    return result;
}
```

Ejemplo 1.8

El método *hashCode()* puede dar lugar a colisiones y para determinar si son iguales o no se utiliza el método *equals()*, por este motivo siempre que se necesite se deben sobrescribir ambos. En este caso concreto no los dará, devuelve un número único entre el '0' y el '52' para cada carta y por ello he decidido utilizarlo en la implementación del método *equals()*.

Volviendo a la ejecución del ejemplo 1.7 vemos el resultado en la ejecución 1.3.

```
22:00:00.000 [main] DEBUG org.poker.api.core.Main - insertamos la carta: AC
22:00:00.001 [main] DEBUG org.poker.api.core.Main - insertamos la carta: 2C
22:00:00.002 [main] DEBUG org.poker.api.core.Main - insertamos la carta: 3C
22:00:00.003 [main] DEBUG org.poker.api.core.Main - la carta: AC ya estaba en el conjunto
```

Ejecución 1.3

Ya tenemos el comportamiento esperado, la clase *Card* ha dado un salto de calidad pero aún faltan algunos detalles para que quede robusta, por ejemplo: el constructor debe recibir siempre parámetros válidos es decir no nulos.

¿Qué ocurriría en el ejemplo 1.7 si agregamos la creación de una carta con un parámetro nulo a la lista de Cartas a insertar?

Veamos el resultado en el ejemplo 1.9:

```
public class Main {
    private static final Logger LOGGER = LoggerFactory.getLogger(Main.class);

    private static void insert(Set<Card> cards, Card card){
        if (!cards.contains(card)){
            LOGGER.debug("insertamos la carta: {}", card);
            cards.add(card);
        } else {
            LOGGER.debug("la carta: {} ya estaba en el conjunto", card);
        }
    }

    public static void main(String[] args) {
        Set<Card> cards = new HashSet<>();
        Card[] cards2Insert = {
            new Card(Card.Suit.CLUB, Card.Rank.ACE),
            new Card(Card.Suit.CLUB, Card.Rank.TWO),
            new Card(Card.Suit.CLUB, Card.Rank.TRHEE),
            new Card(Card.Suit.CLUB, Card.Rank.ACE),
            new Card(Card.Suit.CLUB, null),
        };
        for (Card card : cards2Insert) {
            insert(cards, card);
        }
    }
}
```

```
13:27:25.983 [main] DEBUG org.poker.api.core.Main - insertamos la carta: AC
13:27:25.986 [main] DEBUG org.poker.api.core.Main - insertamos la carta: 2C
13:27:25.986 [main] DEBUG org.poker.api.core.Main - insertamos la carta: 3C
13:27:25.987 [main] DEBUG org.poker.api.core.Main - la carta: AC ya estaba en el conjunto
Exception in thread "main" java.lang.NullPointerException
	at org.poker.api.core.Card.hashCode(Card.java:67)
	at java.util.HashMap.hash(HashMap.java:338)
	at java.util.HashMap.containsKey(HashMap.java:595)
	at java.util.HashSet.contains(HashSet.java:203)
	at org.poker.api.core.Main.insert(Main.java:34)
	at org.poker.api.core.Main.main(Main.java:52)
```

Ejemplo 1.9

Está claro que el ejemplo debería devolver una excepción, pero la cuestión es que ésta no se está produciendo en el lugar más adecuado, permitiendo la posibilidad de que la excepción aparezca en cualquier otro lugar del código bien implementado, cuando este inserte cartas en una colección recibidas como parámetro y sin saber que pueden estar mal creadas.

Vamos a modificar el constructor de la clase *Card* para impedir esta situación, en el ejemplo 1.10 tenemos el resultado.

```
public Card(Suit suit, Rank rank) {
    if (suit == null) {
        throw new IllegalArgumentException("suit no puede tener un valor nulo");
    }
    if (rank == null) {
        throw new IllegalArgumentException("rank no puede tener un valor nulo");
    }
    this.suit = suit;
    this.rank = rank;
}
```

Ejemplo 1.10

Si volvemos a ejecutar el ejemplo 1.9 obtenemos el siguiente resultado en la ejecución 1.4.

```
Exception in thread "main" java.lang.IllegalArgumentException: rank no puede tener un
valor nulo
      at org.poker.api.core.Card.<init>(Card.java:49)
      at org.poker.api.core.Main.main(Main.java:44)
```

Ejecución 1.4

Podemos comprobar como ahora tenemos una excepción indicando claramente cuál es el origen del problema y está se lanza en el lugar apropiado para solventarlo.

Hasta este aquí hemos hecho los cambios necesarios para conseguir una implementación robusta de la clase *Card*, hemos realizado unos ejemplos para verificar su comportamiento, pero es una verificación muy escasa con unas pruebas bastante precarias que necesitan de una revisión manual para determinar si se han pasado o no.

Vamos a implementar test unitarios para la clase *Card* utilizando un framework de test, he seleccionado el más popular: JUnit, en su versión 4.11, otra posibilidad sería utilizar TestNG.

JUnit es un framework que permite ejecutar casos de prueba de una clase, esto ocurre en un entorno controlado, donde se evalúa si para una determinada entrada se obtiene el resultado esperado tras la ejecución del caso de prueba. Una métrica de calidad de software consiste en medir el porcentaje de código de una clase que se ejecuta durante los test unitarios, a este concepto se le denomina cobertura del código.

Diseñar software de modo que permita realizar test unitarios de forma sencilla cubriendo la práctica totalidad del código final no es una tarea nada fácil, pero utilizando patrones como la inyección de dependencias o la inversión de control resulta mucho más sencilla.

Definamos la clase de test unitarios para *Card*, por convención se llamará *CardTest* y se ubicara en el mismo paquete que la clase *Card*, pero deberá estar separada en otra estructura de directorios. Los test son independientes entre sí y se ejecutan sobre instancias distintas de la clase *CardTest*.

En el ejemplo 1.11 hay tres test unitarios para el constructor, en el código de los test debe utilizar la salida estándar el lugar del logger para evitar agregar otros componentes al test. Los tienen que ser útiles y legibles, sobreponiendo la legibilidad sobre el rendimiento.

```
public class CardTest {

    @Test
    public void testContructor() {
        System.out.println("card()");
        Card.Suit expSuit = Card.Suit.CLUB;
        Card.Rank expRank = Card.Rank.TWO;
        Card instance = new Card(expSuit, expRank);
        Card.Suit suitResult = instance.getSuit();
        assertEquals(expSuit, suitResult);
        Card.Rank rankResult = instance.getRank();
        assertEquals(expRank, rankResult);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testContructorSuitNull() {
        System.out.println("card(SuitNull)");
        Card.Suit expSuit = null;
        Card.Rank expRank = Card.Rank.TWO;
        Card instance = new Card(expSuit, expRank);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testContructorRankNull() {
        System.out.println("card(RankNull)");
        Card.Suit expSuit = Card.Suit.CLUB;
        Card.Rank expRank = null;
        Card instance = new Card(expSuit, expRank);
    }
}
```

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.poker.api.core.CardTest
card(SuitNull)
card()
card(RankNull)
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec

Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

Ejemplo 1.11

Revisando los test vemos que en el primer test se espera un funcionamiento adecuado del constructor, los dos siguientes test esperan una excepción, con lo cual para que Junit de los test por buenos en el primer caso se deben cumplir los *assert* y en el segundo y tercero se debe lanzar la excepción *IllegalArgumentException*.

Los assert mencionados serán los que nos proporciona el framework JUnit, no los del lenguaje, siempre que hable de assert me referiré a los primeros a no ser que lo diga expresamente.

Vamos a incluir funcionalidad para realizar test exhaustivos sobre los métodos hashCode y equals en el ejemplo 1.12a y 1.12b.

```java
import java.util.HashSet;
import java.util.Set;
import org.junit.Test;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.hasItem;
import static org.hamcrest.CoreMatchers.not;

public class CardTest {
    ...
    private static Card[] getAllCards() {
        Card[] result = new Card[Card.Suit.values().length * Card.Rank.values().length];
        int i = 0;
        for (Card.Suit suit : Card.Suit.values()) {
            for (Card.Rank rank : Card.Rank.values()) {
                Card c = new Card(suit, rank);
                result[i] = c;
                i++;
            }
        }
        return result;
    }

    @Test
    public void testHashCode() {
        System.out.println("hashCode");
        Card[] allCards = getAllCards();
        Set<Integer> hashCodes = new HashSet<>(allCards.length);
        for (Card card : allCards) {
            assertThat(hashCodes, not(hasItem(card.hashCode())));
        }
    }

    @Test
    public void testEqualsOtherObjects() {
        System.out.println("equalsOtherObjects");
        Card card = new Card(Card.Suit.CLUB, Card.Rank.ACE);
        assertNotEquals("card: " + card + " != null", card, null);
        assertNotEquals("card: " + card + " != 0", card, 0);
        assertNotEquals("card: " + card + " != \"2C\"", card, "2C");
    }
    ...
```

Ejemplo 1.12a

```
    ...
    @Test
    public void testEquals() {
        System.out.println("equals");
        int i = 0;
        for (Card card0 : getAllCards()) {
            int j = 0;
            for (Card card1 : getAllCards()) {
                if (i == j) {
                    assertEquals(card0, card1);
                }
                j++;
            }
            i++;
        }
    }

    @Test
    public void testEqualsDistinct() {
        System.out.println("equals distinct");
        int i = 0;
        for (Card card0 : getAllCards()) {
            int j = 0;
            for (Card card1 : getAllCards()) {
                if (i != j) {
                    assertNotEquals(card0, card1);
                }
                j++;
            }
            i++;
        }
    }
}
```

Ejemplo 1.12b

En primer lugar creamos un método estático que devuelve todas las cartas, JUnit sabe que no es un test ya que no tiene la anotación *@Test*, el resto de métodos como si son test la deben tener precediendo su definición.

El *testEquals* recorre un *array* con todas las cartas y para cada una de ellas vuelve a recorrer la lista completa para verificar que solo es igual a la carta que ocupa su misma posición o sea a sí misma.

El *testHashCode* calcula los valores *hashCode* de todas las cartas y comprueba que ninguno se repite, utilizando *asserts* más expresivos, esto es comprobando que en el conjunto de *hash* (*hashCodes*) no está el valor de la carta que estamos comprobando y después lo incluimos.

El resultado de la ejecución completa de los test es el que se puede ver en la ejecución 1.5.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.poker.api.core.CardTest
equals distinct
card(SuitNull)
card()
equalsOtherObjects
hashCode
card(RankNull)
equals
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 sec

Results :

Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
```

Ejecución 1.5

JUnit ha realizado el trabajo de comprobar los test y ha verificado que todos han dado el resultado esperado, de modo que en este punto ya tenemos un mecanismo de comprobación robusto y automatizado.



Sobre los assert en los test, hay un regla muy estricta que dice: “un assert por test”, personalmente prefiero otra: “comprobar en un test un único comportamiento”, sin importar el número de assert.

En la metodología de pruebas unitarias se manejan una serie de características que los test deben cumplir, como por ejemplo deben de ser independientes, repetibles; también hay otras características deseables como que sean automatizables, tanto en su ejecución como comprobación, que cubran la mayor parte de código y que la relación legibilidad/rendimiento se decante del lado de la legibilidad.

Hay un planteamiento general que implica no mezclar la lógica del código con el de los test, ya que en el proceso de quitar este código se puede dar lugar a errores y a comportamientos indeseados. La primera aproximación fue la inclusión de assert como parte del propio lenguaje en el código que permitían ser eliminados de forma automática, estas aserciones eran una simple comprobación de un único valor booleano, con las limitaciones que ello posee.

Con el tiempo se ha ido evolucionando, se ha empezado a emplear frameworks de pruebas unitarias más elaborados, con más potencia y expresividad, se han llegado a emplear lenguajes específicos

para este fin.

La tendencia ha ido desde separar en distintos ficheros a los test de la lógica del código hasta llegar a separar en directorios diferentes los ficheros de los fuentes del código principal del de los test. En cuanto a la expresividad, la evolución ha sido contundente:

Con los assert:

1. Assert de 1ª generación, booleanos: AssertTrue/AssertFalse.
2. Assert de 2ª generación, tipos primitivos, objetos, arrays: AssertEquals, AssertNull.
3. Assert de 3ª generación: AssertThat + Matchers (Hamcrest o AssertJ).

En cuanto al código:

1. Java con Herencia (con sobre-escritura de métodos).
2. Java Anotaciones (Clases simples más anotaciones).
3. Java | Scala | Groovy + DSL.

Las estrategias de pruebas utilizando *Mocks* han marcado un punto de inflexión importante, no sólo para las pruebas, invitan a hacer diseños más sencillos, piezas mejor desacopladas, que nos permitan testar comportamientos sustituyendo las dependencias con comportamiento real por mocks con comportamientos simulado, listas para provocar el efecto deseado en el módulo que queremos probar, como si de un laboratorio se tratase. Esto es posible hacerlo de forma sencilla con una cantidad mínima de código gracias a frameworks como EasyMock, JMock, Mockito, JMockit o PowerMock.

La irrupción de BDD (Behaviour-Driven Development) con tecnologías como JBehave, Cucumber, Instinct, JDave o BeanSpec ha supuesto otra nueva vuelta de tuerca, tratando de organizar los test para dotarlos de una semántica más natural, con cláusulas como Given, When y Then.

Vemos cómo se ha evolucionado desde el ***diseño por contrato*** hasta los ecosistemas de herramientas y metodologías que tenemos hoy en día (y en constante mejora) sin renunciar a la esencia: Asegurar la calidad y mejorar la mantenibilidad del software.

Aunque el desarrollo está en una fase temprana, es muy común (y deseable) poder hacer modificaciones sobre lo ya desarrollado para mejorarlo, los test nos ayudan a afrontar con más garantías estos cambios.

Veamos lo con un ejemplo, ahora queremos hacer una modificación sobre la clase *Card*, en concreto sobre el enumerado *Suit*, como los enumerados son extremadamente potentes y apenas los hemos exprimido, vamos a realizar una modificación para ganar un poco más en expresividad, vamos a asignar un carácter a cada palo, de modo que el método *toString()* de *Card* cuando la carta sea el dos de corazones devuelva 2♥.

Lo vemos en el ejemplo 1.13.

```
public final class Card {
    ...
    public static enum Suit {
        SPADE('♠'), HEART('♥'), DIAMOND('♦'), CLUB('♣');
        private Suit(char c) {
            this.c = c;
        }
        private final char c;
    }
    ...
}
```

Ejemplo 1.13

Comprobamos como podemos definir constructores, atributos e incluso métodos en un enumerado, lo que no tenemos disponible es la herencia sin embargo se pueden implementar interfaces. Para utilizar el carácter que hemos almacenado en cada elemento del enumerado, lo haremos desde el método *toString* de la clase *Card*.

```
public final class Card {
    ...
    @Override
    public String toString() {
        int r = rank.ordinal();
        return STRING_RANK_CARDS.substring(r, r + 1) + suit.c;
    }
    ...
}
```

Ejemplo 1.14

Después de estos cambios, podría haber introducido algún error en el código, para cerciorarnos de que todo sigue funcionando como debería basta con volver a ejecutar los test que tenemos desarrollados.

En la ejecución 1.6 vemos el resultado.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.poker.api.core.CardTest
equals distinct
card(SuitNull)
card()
equalsOtherObjects
hashCode
card(RankNull)
equals
Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec

Results :

Tests run: 7, Failures: 0, Errors: 0, Skipped: 0
```

Ejecución 1.6

Después de ver como los test unitarios ayudan al mantenimiento y permiten asegurar el funcionamiento de elementos verificados, ponemos fin a este primer capítulo.

Capítulo 2: Algoritmos iniciales

En este capítulo vamos a desarrollar varios algoritmos: para la evaluación de manos de Póquer, gestionar una baraja de cartas, cálculo de combinaciones o una clase de utilidades.

Comenzaremos por la implementación la clase baraja, en el capítulo anterior implementamos un test que necesitaba obtener todas las cartas, como esa funcionalidad se puede necesitar en otro lugar, vamos a refactorizar el código, la extraemos de la clase de test y la agregamos a *Deck*, en el ejemplo 2.1 podemos verlo.

```
public class Deck {

    private final List<Card> cards;
    private int index = 0;

    public Deck() {
        this.cards = getAllCards();
    }

    public Card obtainCard() {
        Card result = null;
        if (index < cards.size()){
            result = cards.get(index);
            index++;
        }
        return result;
    }

    public void shuffle(){
        index = 0;
        Collections.shuffle(cards);
    }

    public static List<Card> getAllCards() {
        int numCards = Card.Suit.values().length * Card.Rank.values().length;
        List<Card> result = new ArrayList<>(numCards);
        for (Card.Suit suit : Card.Suit.values()) {
            for (Card.Rank rank : Card.Rank.values()) {
                result.add(new Card(suit, rank));
            }
        }
        return result;
    }
}
```

Ejemplo 2.1

Para el método *obtainCard* utilizamos un índice que nos indica la siguiente carta que debe devolver, el método *shuffle* mezcla las cartas y vuelve a posicionar el índice de carta actual en la primera posición.

Debemos definir la interfaz del algoritmo de evaluación (Ejemplo 2.2), definir interfaces y utilizarlas en el tipo de los objetos en lugar de clases concretas facilitan la adaptabilidad del código. El algoritmo de evaluación de cartas va a tener un único método, en el que reciba un *array* de cinco cartas y devolverá un valor numérico, en este número pretendemos codificar el valor de cada posible mano.

```
public interface IHandEvaluator {
     public int eval(Card[] cards);
}
```

Ejemplo 2.2

Como hay que realizar comprobaciones de entrada de datos, vamos a refactorizar el código de la clase *Card* y crear una utilidad que permita lanzar excepciones si no se reciben los argumentos adecuados: *ExceptionUtil*, el resultado en el ejemplo 2.3.

```
public final class ExceptionUtil {

    public static final String NULL_ERR_MSG = "El argumento {0} no puede ser nulo.";
    public static final String LENGTH_ERR_MSG =
              "El argumento {0} no puede ser nulo y debe tener una longitud de {1}.";

    private ExceptionUtil() {
    }

    public static void checkNullArgument(Object o, String name) {
        if (o == null) {
            throw new IllegalArg...(MessageFormat.format(NULL_ERR_MSG , name));
        }
    }

    public static <T> void checkArrayLengthArgument(T[] a, String name, int l) {
        if (a == null || a.length != l) {
            throw new IllegalArg...(MessageFormat.format(LENGTH_ERR_MSG, name, l));
        }
    }

    public static void checkArgument(boolean throwEx, String msg, Object... args) {
        if (throwEx) {
            throw new IllegalArgumentException(MessageFormat.format(msg, args));
        }
    }
}
```

Ejemplo 2.3



Las clases con métodos estáticos pueden presentar un problema, suelen complicar la implementación de test unitarios (powermock ayuda a resolver este problema), sobre todo si internamente interactúan con otros componentes. En este caso la funcionalidad está acotada y el valor devuelto no varía nunca.

En este punto vamos a describir como pretendemos que sea la implementación de la interfaz *IhandEvaluator*, en primer lugar habrá que asegurarse que la entrada recibida es adecuada, es decir validar que el array de cartas no es nulo, que tiene cinco elementos no nulos y que éstas no se repitan. Después vamos a contar el número de elementos de cada palo y el número de elementos con cada rango. Como el As puede hacer las veces de As y de 1 en la escalera habrá que tenerlo en cuenta.

Para obtener un método sencillo de evaluación de manos para el desempate en caso de coincidir el tipo de mano, construimos unos índices, donde se almacenan el rango de la carta y el número de cartas de ese rango, ordenando primero por número de cartas y después por rango. Veamos la tabla 2.1:

Cartas desordenadas	Índices calculados	Tipo de mano
9♣ K♠ 3♦ 4♥ 5♠	`{{K,1},{9,1},{5,1},{4,1},{3,1}}`	Carta más alta
4♣ K♥ 3♠ 4♠ 5♦	`{{4,2},{K,1},{5,1},{3,1}}`	Pareja de 4
4♣ K♥ 4♥ 4♠ 4♦	`{{4,4},{K,1}}`	Póquer de 4
4♣ K♥ K♦ 4♠ A♦	`{{K,2},{4,2},{A,1}}`	Dobles parejas de K y 4
4♣ 4♥ A♠ 4♦ A♦	`{{4,3},{A,2}}`	Full house de 4 y Ases

Tabla 2.1

Podemos ver que el palo no afecta para los índices. ¿Qué queremos que ocurra para los casos especiales? pues queremos que el resultado quede como en la tabla 2.2.

Cartas desordenadas	Índices calculados	Tipo de mano
J♣ K♠ T♦ Q♥ A♠	`{{A,1},{K,1},{Q,1},{J,1},{T,1}}`	Escalera al As
4♦ A♦ 3♦ 2♦ 5♦	`{{5,1},{4,1},{3,1},{2,1},{A,1}}`	Escalera de color al 5

Tabla 2.2

En la primera posición de cada par recogido en los índices, donde se indica el rango se utilizará el valor del ordinal de ese enumerado.

La comprobación de si todas las cartas son del mismo palo es sencilla, basta con mirar las cartas que hay del mismo palo que la primera, y para saber si hay escalera hay que comprobar si hay cinco

cartas consecutivas y además tener en cuenta el caso especial de la escalera al 5.

Ahora queda ver como codificamos el resultado de la evaluación, como el primer criterio es el tipo de categoría utilizaremos los ordinales de los tipos de manos, después utilizando los índices obtenemos los rangos de las cartas por orden de relevancia. Veamos unos ejemplos en la tabla 2.3.

Cartas desordenadas	Índices calculados	Cartas por relevancia	Valores
9♣ K♠ 3♦ 4♥ 5♠	{{K,1},{9,1},{5,1},{4,1},{3,1}}	{K, 9, 5, 4, 3}	{11,7,3,2,1}
4♣ K♥ 3♠ 4♠ 5♦	{{4,2},{K,1},{5,1},{3,1}}	{4, 4, K, 5, 3}	{2,2,11,3,1}
4♣ K♥ 4♥ 4♠ 4♦	{{4,4},{K,1}}	{4, 4, 4, 4, K}	{2,2,2,2,11}
4♣ K♥ K♦ 4♠ A♦	{{K,2},{4,2},{A,1}}	{K, K, 4, 4, A}	{11,11,2,2,12}
4♣ 4♥ A♠ 4♦ A♦	{{4,3},{A,2}}	{4, 4, 4, A, A}	{2,2,2,12,12}
J♣ K♠ T♦ Q♥ A♠	{{A,1},{K,1},{Q,1},{J,1},{T,1}}	{A, K, Q, J, T}	{12,11,10,9,8}
4♦ A♦ 3♦ 2♦ 5♦	{{5,1},{4,1},{3,1},{2,1},{A,1}}	{5, 4, 3, 2, A}	{3,2,1,0,12}

Tabla 2.3

Utilizaremos la siguiente formula:

```
Tipo*Base⁵+Carta5*Base⁴+Carta4*Base³+Carta3*Base²+Carta2*Base+Carta1
```

Dónde:

```
Base: (Mayor ordinal de una carta: Rank.ACE.getOrdinal() → 12) + 1 → 13
Tipo: El valor ordinal del tipo de categoría.
Carta5: El ordinal de la carta más relevante.
Carta4: El ordinal de la segunda carta más relevante.
Carta3: El ordinal de la tercera carta más relevante.
Carta2: El ordinal de la segunda carta menos relevante.
Carta1: El ordinal de la carta menos relevante.
```

En la tabla 2.4 vemos los resultados de las manos de la tabla 2.3.

Cartas desordenadas	Mano	Valor tipo mano	Valores de las cartas	Valor de la mano
9♣ K♠ 3♦ 4♥ 5♠	La carta más alta	0	{11,7,3,2,1}	330.084
4♣ K♥ 3♠ 4♠ 5♦	Pareja de 4	1	{2,2,11,3,1}	434.708
4♣ K♥ 4♥ 4♠ 4♦	Póquer de 4	7	{2,2,2,2,11}	2.660.942
4♣ K♥ K♦ 4♠ A♦	Dobles Parejas de Ks y 4	2	{11,11,2,2,12}	1.081.300
4♣ 4♥ A♠ 4♦ A♦	Full House de 4 y Ases	6	{2,2,2,12,12}	2.289.780
J♣ K♠ T♦ Q♥ A♠	Escalera al As	4	{12,11,10,9,8}	1.853.886
4♦ A♦ 3♦ 2♦ 5♦	Escalera de color al 5	8	{3,2,1,0,12}	3.060.602

Tabla 2.4

En los ejemplos 2.4a y 2.4b está el código del evaluador *HandEvaluator*, implementa la interfaz *IHandEvaluator*.

```
public final class HandEvaluator implements IHandEvaluator {

    private static final int ENCODE_BASE = Card.Rank.ACE.ordinal() + 1;
    private static final int INDEXES_LENGTH = 2;
    private static final int RANK_INDEX = 0;
    private static final int REPEATS_INDEX = 1;
    private static final Type[][] MATRIX_TYPES = {
        {Type.HIGH_CARD}, {Type.ONE_PAIR, Type.TWO_PAIR},
        {Type.THREE_OF_A_KIND, Type.FULL_HOUSE},{Type.FOUR_OF_A_KIND}
    };
    private final int[][] indexes = new int[Hands.CARDS][INDEXES_LENGTH];
    private final int[] ranks = new int[ENCODE_BASE];
    private final int[] suits = new int[Card.Suit.values().length];
    private boolean isStraight = false;
    private boolean isFlush = false;

    @Override
    public int eval(Card[] cards) {
        ExceptionUtil.checkArrayLengthArgument(cards, "cards", Hands.CARDS);
        isFlush = false;
        Arrays.fill(suits, 0);
        Arrays.fill(ranks, 0);
        int index = 0;
        Set<Card> previousCards = new HashSet<>(Hands.CARDS);
        for (Card card : cards) {
            ExceptionUtil.checkNullArgument(card, "card[" + (index++) + "]");
            ExceptionUtil.checkArgument(previousCards.contains(card),
                                                     "La carta {} está repetida.", card);
            previousCards.add(card);
            ranks[card.getRank().ordinal()]++;
            suits[card.getSuit().ordinal()]++;
        }
        isFlush = suits[cards[0].getSuit().ordinal()] == Hands.CARDS;
        isStraight = false;
        int straightCounter = 0;
        int j = 0;
        for (int i = ranks.length - 1; i >= 0; i--) {
            if (ranks[i] > 0) {
                straightCounter++;
                isStraight = straightCounter == Hands.CARDS;
                indexes[j][RANK_INDEX] = i;
                indexes[j][REPEATS_INDEX] = ranks[i];
                upIndex(j++);
            } else {
                straightCounter = 0;
            }
        }
        isStraight = isStraight || checkStraight5toAce(straightCounter);
        return calculateHandValue();
    }
    ...
```

Ejemplo 2.4a

```
    ...
    // Actualiza el orden de los pares en los índices.
    private void upIndex(int i) {
        int k = i;
        while (k > 0 && indexes[k - 1][REPEATS_INDEX] < indexes[k][REPEATS_INDEX]) {
            int[] temp = indexes[k - 1];
            indexes[k - 1] = indexes[k];
            indexes[k] = temp;
            k--;
        }
    }

    private boolean checkStraight5toAce(int straightCntr) {
        boolean straight5toAce = false;
        // Evalúa si se trata del caso especial
        if (ranks[Card.Rank.ACE.ordinal()] == 1 && straightCntr == Hands.CARDS - 1) {
            // Si es el caso especial hay que reorganizar los índices
            straight5toAce = true;
            for (int i = 1; i < indexes.length; i++) {
                indexes[i - 1][RANK_INDEX] = indexes[i][RANK_INDEX];
            }
            indexes[indexes.length - 1][RANK_INDEX] = Card.Rank.ACE.ordinal();
        }
        return straight5toAce;
    }

    private int calculateHandValue() {
        Type type;
        if (isStraight) {
            type = isFlush ? Type.STRAIGHT_FLUSH : Type.STRAIGHT;
        } else if (isFlush) {
            type = Type.FLUSH;
        } else {
            type = MATRIX_TYPES[indexes[0][REPEATS_INDEX] -1][indexes[1][REPEATS_INDEX] -1];
        }
        return encodeValue(type, indexes);
    }

    private static int encodeValue(Type type, int[][] indexes) {
        int result = type.ordinal();
        int i = 0;
        int j = 0;
        while (j < Hands.CARDS) {
            for (int k = 0; k < indexes[i][REPEATS_INDEX]; k++) {
                result = result * ENCODE_BASE + indexes[i][RANK_INDEX];
                j++;
            }
            i++;
        }
        return result;
    }
}
```

Ejemplo 2.4b

La utilidad *ExceptionUtil* validará los parámetros de entrada del algoritmo de evaluación de manos.

Hemos definido una matriz de constantes, contiene el valor pre-calculado de los índices, esto nos va a permitir devolver el tipo de categoría de forma automática en el método *calculateHandValue*.

Con esta matriz, los índices y los dos indicadores (escalera y color) se determina con facilidad la categoría de la mano, como la categoría más alta es la de la escalera de color es la primera que se comprueba junto con la escalera sin color, más adelante el color y finalmente en función de la matriz y los dos índices de mayor peso se determinan el resto de categorías.

Índices:

Todos – Dos primeros
{1, 1, 1, 1, 1} → Carta más alta {1, 1}.
{2, 1, 1, 1, 0} → Pareja {2, 1}.
{2, 2, 1, 0, 0} → Dobles parejas {2, 2}.
{3, 1, 1, 0, 0} → Trío {3, 1}.
{3, 2, 0, 0, 0} → Full {3, 2}.
{4, 1, 0, 0, 0} → Poker {4, 1}

Matriz:

Índice 1\ Índice 2	Índice 2 = 1	Índice 2 = 2
Índice 1 = 1	Carta más alta {1, 1}	-
Índice 1 = 2	Pareja {2, 1}	Dobles parejas {2, 2}
Índice 1 = 3	Trío {3, 1}	Full {3, 2}
Índice 1 = 4	Poker {4, 1}	-

Hasta aquí damos por finalizado el algoritmo de evaluación, ahora vamos a agregar algunos test unitarios. Pero a diferencia de lo que hemos visto hasta ahora vamos a utilizar un modo distinto de enfocarlo, en el capítulo anterior hablamos de DSL (Domain Specific Language) y de BDD (Behaviour-Driven Development) y mencionamos las tecnologías JBehave, Cucumber, Instinct, JDave y BeanSpec.

Ahora toca el turno de explorar un poco está metodología. De todas las tecnologías mencionadas vamos a escoger Cucumber, es la más popular y junto a su lenguaje de dominio específico es la que me resulta más interesante.

En primer lugar decir que Cucumber utiliza un framework de pruebas unitarios JUnit o Testng, en el caso de integrarse con JUnit se apoya en anotaciones para indicar que Cucumber debe inicializarse y como queremos configurarlo.

La clase *CukesRunnerTest* es la que vamos a utilizar para lanzar Cucumber y configurarlo, en el ejemplo 2.5 vemos el resultado.

```
RunWith(Cucumber.class)
@CucumberOptions(features = "src/test/resources")
public class CukesRunnerTest {
}
```

Ejemplo 2.5

Lo primero que apreciamos es que la clase está completamente vacía, sólo contiene dos anotaciones, una para que JUnit sepa que tiene que iniciar Cucumber y la otra para decir donde están definidos los ficheros de características de nuestro código a probar. El orden lógico sería construir nuestros ficheros de características y después crear el código de los pasos a seguir con la ayuda del framework Cucumber.

Nuestro fichero de características se llamará HandEvaluator.feature, la extensión es importante, además que debe situarse en uno de los subdirectorios que hemos indicado a la clase del test. En el ejemplo 2.6 podemos ver cómo queda el fichero.

```
Feature: Evaluación de manos de poker
  Queremos saber lo fuerte que es una mano de poker.

  Scenario Outline: Evaluación de dos manos de poker
    Given un HandEvaluator
    When calculamos la comparacion entre <mano0> y <mano1>
    Then el resultado esperado es <resultado>

    Examples: comparaciones con escaleras de color
      | mano0          | mano1          | resultado |
      | A♣ K♣ T♣ J♣ Q♣ | Q♣ K♣ T♣ J♣ A♣ | iguales   |
      | A♣ K♣ T♣ J♣ Q♣ | 9♣ K♣ T♣ J♣ Q♣ | mano0     |
      | A♣ 5♣ 4♣ 3♣ 2♣ | 6♣ 5♣ 4♣ 3♣ 2♣ | mano1     |
      | A♥ 2♥ 4♥ 3♥ 5♥ | A♣ 5♣ 4♣ 3♣ 2♣ | iguales   |

    Examples: comparaciones con poker
      | mano0          | mano1          | resultado |
      | A♣ A♥ A♦ A♠ K♠ | A♣ A♥ A♦ A♠ K♠ | iguales   |
      | A♣ A♥ A♦ A♠ K♠ | A♣ A♥ A♦ A♠ Q♠ | mano0     |
      | A♣ A♥ A♦ A♠ J♠ | A♣ A♥ A♦ A♠ Q♠ | mano1     |
      | T♣ T♥ T♦ T♠ 9♠ | T♣ T♥ T♦ T♠ 9♥ | iguales   |
      | T♣ T♥ T♦ T♠ 9♠ | 9♣ 9♥ 9♦ 9♠ A♥ | mano0     |
      | K♣ K♥ K♦ K♠ 9♠ | Q♣ Q♥ Q♦ Q♠ A♥ | mano0     |
```

Ejemplo 2.6

Este no es el ejemplo más sencillo, pero lo utilizaremos como guía, contiene las palabras reservadas en azul y las variables de los test en verde. Tiene la característica definida al principio con lenguaje natural, luego tiene los escenarios posibles, en el ejemplo únicamente se define uno, en un ejemplo sencillo utilizaría sólo la palabra ***Scenario*** pero como vamos a tener tablas con ejemplos es necesario utilizar ***Scenario Outline***, luego vienen las tres clausulas: given, when y then donde se define con lenguaje natural el test, si utilizamos las palabras en castellano la frase sería:

```
Dado un HandEvaluator
Cuando calculamos la comparacion entre <mano0> y <mano1>
Entonces el resultado esperado es <resultado>
```

Después tenemos un par de tablas con ejemplos y resultados esperados, ahora falta la unión entre la historia que hemos definido y la funcionalidad que queremos probar. Este punto lo realizaremos en la clase *HandEvaluatorSteps*, que podemos ver en el ejemplo 2.7.

```
public class HandEvaluatorSteps {
    private static final String[] VALORES = {"mano0", "iguales", "mano1"};
    private IHandEvaluator handEvaluator;
    private String resultado;


    @Given("^un IHandEvaluator$")
    public void un_IHandEvaluator() throws Throwable {
        handEvaluator = new HandEvaluator();
    }

    @When("^calculamos la comparacion entre (.*) y (.*)$")
    public void calculamos_la_comparacion(String h0, String h1) throws Throwable {
        int evalhand0 = handEvaluator.eval(fromString2Cards(hand0));
        int evalhand1 = handEvaluator.eval(fromString2Cards(hand1));
        int diferencia = evalhand1 - evalhand0;
        if (diferencia != 0) {
            diferencia = Math.abs(diferencia) / diferencia;
        }
        resultado = VALORES[diferencia + 1];
    }

    @Then("^el resultado esperado es (.*)$")
    public void el_resultado_esperado_es(String expResult) throws Throwable {
        assertEquals(expResult, resultado);
    }
}
```

Ejemplo 2.7

Nos encontramos con anotaciones en los métodos (@Given @When y @Then), con un parámetro de tipo String, y en él aparece una expresión regular, de la cual se pueden extraer parámetros, como ocurre en este caso con los métodos anotados con @When o @Then.

Los métodos se deben corresponder con los definidos en fichero de características HandEvaluator.feature y los parámetros recibidos deben poder extraerse de la expresión regular. Ponemos fin al evaluador para continuar con la combinatoria.

Vamos a desarrollar un algoritmo que permita calcular todas las combinaciones posibles que hay en la selección de un subgrupo de tamaño fijo sobre un conjunto de total de elementos, en otras palabras, se trata de obtener cada una de las combinaciones que hay en cada elemento del triángulo de Pascal:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
1   9  36  84  126  126  84  36   9   1
```

Uno de los conjuntos de combinaciones que nos interesan son las de la fila 7, sexto elemento, el que tiene el valor de 21. Ya que si en la mesa hay cinco cartas comunitarias y cada jugador tiene dos, hay 21 combinaciones posibles de cinco cartas en ese conjunto de cartas, habría que evaluarlas todas para quedarse con la mejor, ya que esa será la mano del jugador. Por entender de donde salen estas combinaciones vamos a desarrollarlas:

- Cartas comunitarias: C1, C2, C3, C4 y C5.
- Cartas del Jugador: C6 y C7.

Combinaciones posibles de cinco cartas:

1	{C1, C2, C3, C4, C5}	12	{C1, C3, C4, C5, C7}
2	{C1, C2, C3, C4, C6}	13	{C1, C3, C4, C6, C7}
3	{C1, C2, C3, C4, C7}	14	{C1, C3, C5, C6, C7}
4	{C1, C2, C3, C5, C6}	15	{C1, C4, C5, C6, C7}
5	{C1, C2, C3, C5, C7}	16	{C2, C3, C4, C5, C6}
6	{C1, C2, C3, C6, C7}	17	{C2, C3, C4, C5, C7}
7	{C1, C2, C4, C5, C6}	18	{C2, C3, C4, C6, C7}
8	{C1, C2, C4, C5, C7}	19	{C2, C3, C5, C6, C7}
9	{C1, C2, C4, C6, C7}	20	{C2, C4, C5, C6, C7}
10	{C1, C2, C5, C6, C7}	21	{C3, C4, C5, C6, C7}
11	{C1, C3, C4, C5, C6}		

Tabla 2.5

Definamos previamente el interfaz que deseamos a tener, vamos a necesitar un método que nos devuelva el número total de combinaciones, otro que nos devuelva el tamaño del subconjunto, otro más que nos permita empezar de nuevo y luego los dos últimos, uno para saber si hay más combinaciones y otro para devolver los índices de la combinación actual.

En el Ejemplo 2.8 podemos ver como ha quedado la interfaz.

```
public interface ICombinatorial {

    public long combinations();

    public int size();

    public void clear();

    public int[] next(int[] items);

    public boolean hasNext();
}
```

Ejemplo 2.8

En este punto vamos a reflexionar sobre cómo podríamos diseñar unas pruebas para esta interfaz, con la lógica que debe tener el algoritmo de combinaciones descrito anteriormente.

Vamos a definir cuál es la entrada y cuantas combinaciones esperamos tener, la entrada es la dupla de tamaño del subconjunto y la del conjunto total.

Debemos hacer validaciones de la entrada, ya que el subconjunto no puede ser mayor que el conjunto total ni tampoco puede ser menor que uno.

También hay que verificar que las combinaciones (el método *combinations()*) totales son las que esperamos, que el resultado de la invocación a *size()* se corresponde con el tamaño del subconjunto, también que cuando se invoca al método *clear()* se vuelve al principio y finalmente nos quedará la parte más compleja: La verificación de *next* y *hashNext*.

Vamos a iterar en cada una de las combinaciones posibles verificando que mientras no se haya llegado al total de combinaciones el método *hashNext* devuelve el valor verdadero. En cada combinación verificaremos que la actual no se ha repetido y que todos sus valores se encuentran dentro del rango posible.

Una vez haya llegado al final de combinaciones habrá que asegurar que ahora *hashNext* devuelve falso, ejecutar el método de inicialización clear(), y volver a recorrer todas las combinaciones verificando que cada una de ellas es exactamente igual a la que había en la primera iteración.

Vamos a implementar una clase de test unitario para verificar los constructores en el ejemplo 2.9.

```
public class CombinationContructorTest {

    @Test
    public void testConstructor() {
        System.out.println("Combination(2,4)");
        int subItems = 2;
        int items = 4;
        long expectCombinations = 6L;
        Combination instance = new Combination(subItems, items);
        assertEquals(expectCombinations, instance.combinations());
        assertEquals(subItems, instance.size());
    }

    @Test(expected = IllegalArgumentException.class)
    public void testContructorSubItemError() {
        System.out.println("Combination(0,1)");
        int subItems = 0;
        int items = 1;
        Combination instance = new Combination(subItems, items);
    }

    @Test(expected = IllegalArgumentException.class)
    public void testContructorItemError() {
        System.out.println("Combination(5,1)");
        int subItems = 5;
        int items = 1;
        Combination instance = new Combination(subItems, items);
    }
}
```

Ejemplo 2.9

Vamos a necesitar otra clase para test unitarios donde se reciban los parámetros descritos, en el ejemplo 2.10a vemos como se define una clase parametrizable.

```
RunWith(value = Parameterized.class)
public class CombinationTest {

    private final int subItems;
    private final int items;
    private final int combsExpected;

    public CombinationTest(int subItems, int items, int combsExpected) {
        this.subItems = subItems;
        this.items = items;
        this.combsExpected = combsExpected;
    }
    ...
```

Ejemplo 2.10a

JUnit permite definir test parametrizados, para ello es necesario utilizar la anotación @RunWith en la definición de la clase del test, después hay diversos modos de afinar la configuración, es altamente recomendable echar un vistazo al github de junit para conocer las diferentes opciones. En el ejemplo 2.10b se va a definir los parámetros que se van a ejecutar.

```
...
@Parameters
public static Collection<Object[]> data() {
    Object data[][] = {
        {1, 1, 1},
        {1, 2, 2},
        {2, 2, 1},
        {1, 3, 3},
        {2, 3, 3},
        {3, 3, 1},
        {1, 4, 4},
        {2, 4, 6},
        {3, 4, 4},
        {4, 4, 1},
        {1, 5, 5},
        {2, 5, 10},
        {5, 7, 21}
    };
    return Arrays.asList(data);
}
...
```

Ejemplo 2.10b

Pretendemos validar los casos validos más extremos posibles, utilizando valores pequeños, ya que si se utilizan valores muy grandes el tiempo necesario para ejecutar esas pruebas las convierte en inabordables. La primera parte del test que comprueba los invariantes definidos lo tenemos en el ejemplo 2.10c.

```
...
@Test
public void testSize() {
    Combination instance = new Combination(subItems, items);
    int sizeResult = instance.size();
    assertEquals(subItems, sizeResult);
}

@Test
public void testCombinations() {
    System.out.println("combinations: …
    Combination instance = new Combination(subItems, items);
    long result = instance.combinations();
    assertEquals(combinationsExpected, result);
}
...
```

Ejemplo 2.10c

El resto del test lo podemos ver en los ejemplos 2.10d y 2.10e.

```
...
@Test
public void tesHasNextFirst() {
    Combination instance = new Combination(subItems, items);
    assertTrue(instance.hasNext());
}

@Test
public void testHasNextPreLast() {
    Combination instance = new Combination(subItems, items);
    int[] indexes = new int[instance.size()];
    for (int i = 0; i < combinationsExpected - 1; i++) {
        instance.next(indexes);
    }
    assertTrue(instance.hasNext());
}

@Test
public void testHasNextLast() {
    Combination instance = new Combination(subItems, items);
    int[] indexes = new int[instance.size()];
    for (int i = 0; i < combinationsExpected; i++) {
        instance.next(indexes);
    }
    assertFalse(instance.hasNext());
}

@Test
public void testHasNextAfterClear() {
    Combination instance = new Combination(subItems, items);
    instance.clear();
    assertTrue(instance.hasNext());
}

@Test
public void testHasNextAfterClearWithNext() {
    Combination instance = new Combination(subItems, items);
    int indexes[] = new int[subItems];
    instance.next(indexes);
    instance.clear();
    assertTrue(instance.hasNext());
}
@Test
public void testHasNextAfterFullLoopClear() {
    Combination instance = new Combination(subItems, items);
    int indexes[] = new int[subItems];
    for (int i = 0; i < combinationsExpected; i++) {
        instance.next(indexes);
    }
    instance.clear();
    assertTrue(instance.hasNext());
}
...
```

Ejemplo 2.10d

```
    ...
    @Test
    public void testNextItemsRange() {
        Combination instance = new Combination(subItems, items);
        int[] indexes = new int[instance.size()];
        for (int i = 0; i < combinationsExpected; i++) {
            instance.next(indexes);
            assertThat(indexes[0]).isBetween(0, items - 1);
            for (int j = 1; j < subItems; j++) {
                assertThat(indexes[j]).isBetween(indexes[j - 1] + 1, items - 1);
            }
        }
    }
    @Test
    public void testNextDontRepeat() {
        Combination instance = new Combination(subItems, items);
        int results[][] = new int[combinationsExpected][];
        for (int i = 0; i < combinationsExpected; i++) {
            results[i] = new int[subItems];
            instance.next(results[i]);
            for (int k = 0; k < i; k++) {
                assertThat(results[i], not(equalTo(results[k])));
            }
        }
    }
    @Test
    public void testNextIgnored() {
        Combination instance = new Combination(subItems, items);
        int indexes[] = new int[subItems];
        int expIndexes[] = new int[subItems];
        for (int i = 0; i < combinationsExpected; i++) {
            instance.next(indexes);
        }
        Arrays.fill(indexes, -1);
        Arrays.fill(expIndexes, -1);
        instance.next(indexes);
        assertArrayEquals(expIndexes, indexes);
    }
    @Test
    public void testNextAfterClear() {
        Combination instance = new Combination(subItems, items);
        instance.clear();
        int expectIndexes[][] = new int[combinationsExpected][];
        for (int i = 0; i < combinationsExpected; i++) {
            expectIndexes[i] = new int[subItems];
            instance.next(expectIndexes[i]);
        }
        instance.clear();
        int[] indexes = new int[subItems];
        for (int j = 0; j < combinationsExpected - 1; j++) {
            instance.next(indexes);
            assertArrayEquals(expectIndexes[j], indexes);
        }
    }
}
```

Ejemplo 2.10e

En este punto ya debemos empezar a desarrollar el algoritmo de cálculo de Combinaciones, definiremos la clase sin implementar nada más que lo justo para que compile, podemos ver el resultado en el ejemplo 2.11.

```
public class Combination implements ICombinatorial {

    private final int items;
    private final int[] indexes;

    public Combination(int subItems, int items) {
        this.indexes = new int[subItems];
        this.items = items;
    }

    public long combinations() {
        return 0L;
    }

    public int size() {
        return 0;
    }

    public void clear() {
    }

    public int[] next(int[] items) {
        return null;
    }

    public boolean hasNext() {
        return false;
    }
}
```

Ejemplo 2.11

Si lanzamos el primer conjunto de test vemos el resultado en la ejecución 2.1:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.poker.utils.combinatorial.CombinationContructorTest
Combination(5,1)
Combination(0,1)
Combination(2,4)
Tests run: 3, Failures: 3, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec <<<
FAILURE!
```

Ejecución 2.1

Si lanzamos el segundo conjunto de test obtenemos el resultado en la ejecución 2.2.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.util.combinatorial.CombinationTest
combinations: 1, 1, 1
combinations: 1, 2, 2
combinations: 2, 2, 1
combinations: 1, 3, 3
combinations: 2, 3, 3
combinations: 3, 3, 1
combinations: 1, 4, 4
combinations: 2, 4, 6
combinations: 3, 4, 4
combinations: 4, 4, 1
combinations: 1, 5, 5
combinations: 2, 5, 10
combinations: 5, 7, 21
Tests run: 156, Failures: 100, Errors: 13, Skipped: 0, Time elapsed: 0.221 sec <<< FAILURE!

Results :

Failed tests:   testCombinations[0](org.util.combinatorial.CombinationTest): expected:<1> but was:<0>
  testHasNextAfterFullLoopClear[0](org.util.combinatorial.CombinationTest)
  testSize[0](org.util.combinatorial.CombinationTest): expected:<1> but was:<0>
  testHasNextAfterClear[0](org.util.combinatorial.CombinationTest)
  testHasNextPreLast[0](org.util.combinatorial.CombinationTest)
  tesHasNextFirst[0](org.util.combinatorial.CombinationTest)
  testHasNextAfterClearWithNext[0](org.util.combinatorial.CombinationTest)
...

Tests in error:
  testNextItemsRange[0](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[1](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[2](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[3](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[4](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[5](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[6](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[7](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[8](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[9](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[10](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[11](org.util.combinatorial.CombinationTest): 0
  testNextItemsRange[12](org.util.combinatorial.CombinationTest): 0

Tests run: 156, Failures: 100, Errors: 13, Skipped: 0
```

Ejecución 2.2

Ahora retomamos la implementación de la clase *Combination*, en el ejemplo 2.12 vamos a implementar adecuadamente el constructor.

```
    public Combination(int subItems, int items) {
        ExceptionUtil.checkMinValueArgument(subItems, 1, "subItems");
        ExceptionUtil.checkMinValueArgument(items, subItems, "items");
        this.indexes = new int[subItems];
        this.items = items;
    }
```

Ejemplo 2.12

Lanzando el primer conjunto de test, vemos como ya hay algunos test que se empiezan a pasar.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.poker.utils.combinatorial.CombinationContructorTest
Combination(5,1)
Combination(0,1)
Combination(2,4)
Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec <<< FAILURE!

Results :

Failed tests:   testConstructor(o.p.u.c.CombinationContructorTest): expected:<6> but was:<0>
```

Ejecución 2.3

Si seguimos este ciclo donde primero pasamos las pruebas y luego realizamos la implementación de la parte que falla llegamos hasta cubrir toda la funcionalidad. El resultado final de aplicar este proceso lo podemos ver en el ejemplo 2.13a y 2.13b.

```
public class Combination implements ICombinatorial {

    private final int items;
    private final int[] indexes;

    public Combination(int subItems, int items) {
        ExceptionUtil.checkMinValueArgument(subItems, 1, "subItems");
        ExceptionUtil.checkMinValueArgument(items, subItems, "items");
        this.indexes = new int[subItems];
        this.items = items;
        init();
    }

    @Override
    public long combinations() {
        return combinations(indexes.length, items);
    }

    @Override
    public int size() {
        return indexes.length;
    }

    public int getSubItems() {
        return indexes.length;
    }

    public int getItems() {
        return items;
    }
    ...
```

Ejemplo 2.13a

```
    ...
    private boolean hasNext(int index) {
        return indexes[index] + (indexes.length - index) < items;
    }

    private void move(int index) {
        if (hasNext(index)) {
            indexes[index]++;
            int last = indexes[index];
            for (int i = index + 1; i < indexes.length; i++) {
                this.indexes[i] = ++last;
            }
        } else {
            move(index - 1);
        }
    }

    @Override
    public int[] next(int[] items) {
        if (hasNext()) {
            move(indexes.length - 1);
            System.arraycopy(indexes, 0, items, 0, indexes.length);
        }
        return items;
    }

    @Override
    public boolean hasNext() {
        return hasNext(0) || hasNext(indexes.length - 1);
    }

    private void init() {
        int index = indexes.length;
        for (int i = 0; i < indexes.length; i++) {
            this.indexes[i] = i;
        }
        this.indexes[index - 1]--;
    }

    @Override
    public void clear() {
        init();
    }

    public static long combinations(int subItems, int items) {
        long result = 1;
        int sub = Math.max(subItems, items - subItems);
        for (int i = sub + 1; i <= items; i++) {
            result = (result * i) / (i - sub);
        }
        return result;
    }
}
```

Ejemplo 2.13b

El resultado de los test finales lo vemos en la ejecución 2.4.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.util.combinatorial.CombinationTest
combinations: 1, 1, 1
combinations: 1, 2, 2
combinations: 2, 2, 1
combinations: 1, 3, 3
combinations: 2, 3, 3
combinations: 3, 3, 1
combinations: 1, 4, 4
combinations: 2, 4, 6
combinations: 3, 4, 4
combinations: 4, 4, 1
combinations: 1, 5, 5
combinations: 2, 5, 10
combinations: 5, 7, 21
Tests run: 156, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.156 sec
Running org.util.combinatorial.CombinationContructorTest
Combination(5,1)
Combination(0,1)
Combination(2,4)
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec

Results :

Tests run: 159, Failures: 0, Errors: 0, Skipped: 0
```

Ejecución 2.4

La práctica de desarrollo llevada a cabo hasta implementar la clase *Combination* se denomina Test Driven Development (TDD) o desarrollo dirigido por test. Vamos a resumir brevemente los pasos para que queden muy sintetizados:

1.- Definimos los requisitos.
2.- Definimos las pruebas del modo más exhaustivo posible.
3.- Desarrollamos una implementación que compile.
4.- Pasamos las pruebas definidas. Si fallan pasamos al punto 5 y si no, hemos acabado.
5.- Desarrollamos funcionalidad que ha fallado en el test y refactorizamos el código si es preciso.
6.- Volvemos al punto 4.

Esta metodología permite crear código de calidad que cubre los requisitos, es una práctica básica dentro de las metodologías ágiles, no obstante tiene sus limitaciones y sus detractores, ya que no todo el código es testable (p.e. interfaces gráficas de usuario, componentes que interactúan con otro software directamente) y necesita tener los requisitos bien definidos y estables. Mi opinión personal al respecto es que primero habría que conocer bien esta metodología, entender las limitaciones y aplicarla cuando ofrezca más ventajas que inconvenientes que suelen ser en la mayoría de los proyectos que requieren de un alto grado de calidad final donde los requisitos están bien definidos y son estables.

Capítulo 3: Póquer API

A estas alturas hemos visto unas pocas clases, pero vamos a tener que organizar la estructura del código para evitar que se convierta en una jungla. Las clases anteriores las vamos a enmarcar en el API de la aplicación, en concreto en la paquetería *org.poker.api.core;* pero sólo forman una parte del API, faltaría agregar al API las clases que definen la interfaz con los jugadores y con el juego.

Esta nueva funcionalidad la vamos a colocar en la paquetería *org.poker.api.game*. En la estructura final, el API quedaría del modo representado en el diagrama 3.1.

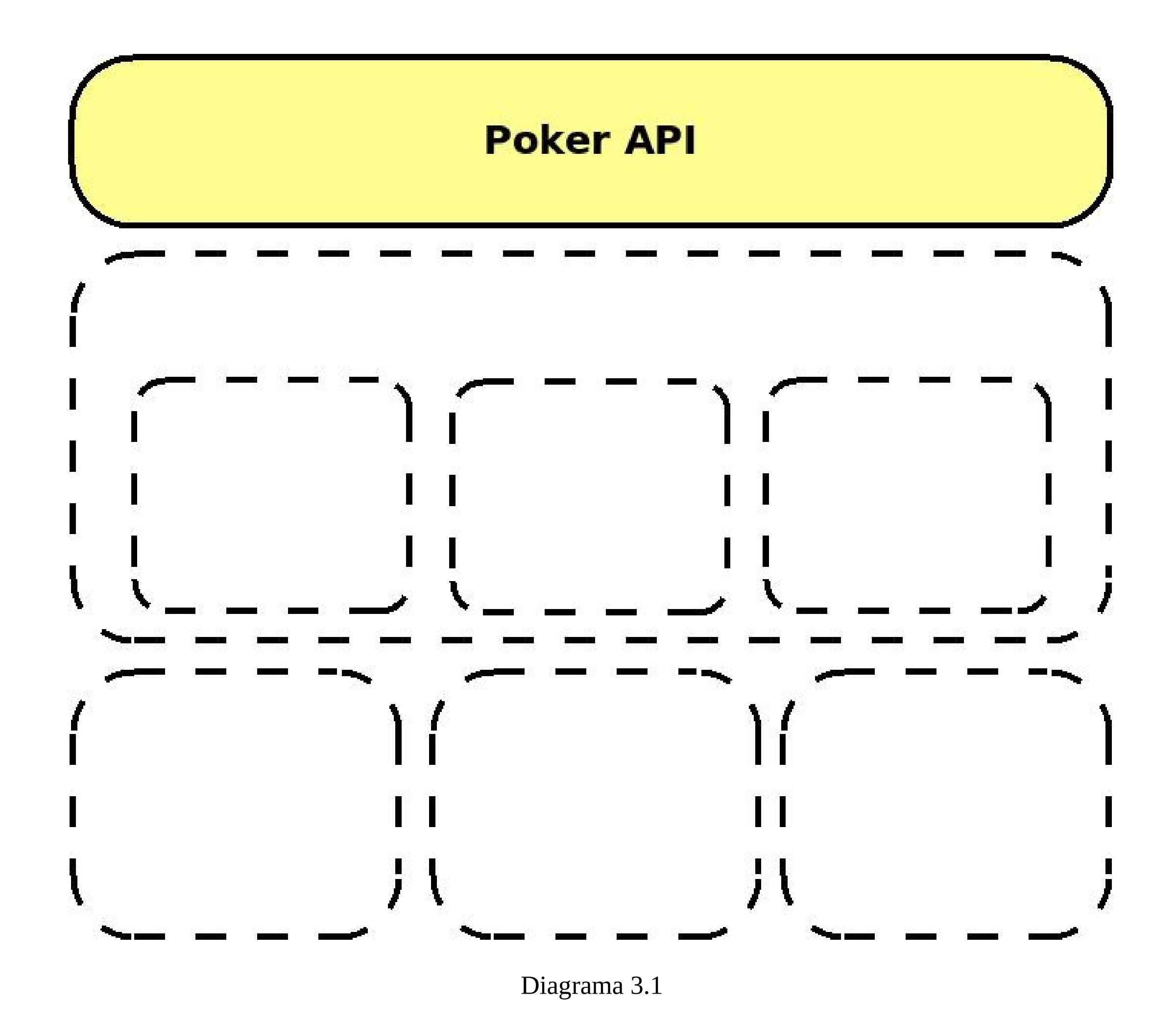


Diagrama 3.1

Para ir completando el API del juego vamos a definir una clase utilidad llamada *TexasHoldEmUtil*, donde podamos reflejar aquellos conceptos del juego de forma global, como por ejemplo el tipo de acciones que puede realizar un jugador, los diferentes estados del juego o el estado de un jugador.

El resultado podemos verlo en el ejemplo 3.1a y 3.1b.

```
public final class TexasHoldEmUtil {

    public static final int MIN_PLAYERS = 2;
    public static final int PLAYER_CARDS = 2;
    public static final int MAX_PLAYERS = 10;
    public static final int COMMUNITY_CARDS = 5;
    private static final Map<BetCommandType, PlayerState> PLAYER_STATE_CONVERSOR =
                                                        buildPlayerStateConversor();

    public enum BetCommandType {
        ERROR, TIMEOUT, FOLD, CHECK, CALL, RAISE, ALL_IN
    }

    public enum GameState {
        PRE_FLOP, FLOP, TURN, RIVER, SHOWDOWN, END
    }

    public enum PlayerState {
        READY(true),
        OUT(false),
        FOLD(false),
        CHECK(true),
        CALL(true),
        RAISE(true),
        ALL_IN(false);

        private final boolean active;

        private PlayerState(boolean isActive) {
            this.active = isActive;
        }

        public boolean isActive() {
            return active;
        }
    }

    private TexasHoldEmUtil() {
    }

    public static PlayerState convert(BetCommandType betCommand) {
        return PLAYER_STATE_CONVERSOR.get(betCommand);
    }
    ...
```

Ejemplo 3.1a

```
    ...
    private static Map<BetCommandType, PlayerState> buildPlayerStateConversor() {
        Map<BetCommandType, PlayerState> result= new EnumMap<>(BetCommandType.class);
        result.put(BetCommandType.FOLD, PlayerState.FOLD);
        result.put(BetCommandType.ALL_IN, PlayerState.ALL_IN);
        result.put(BetCommandType.CALL, PlayerState.CALL);
        result.put(BetCommandType.CHECK, PlayerState.CHECK);
        result.put(BetCommandType.RAISE, PlayerState.RAISE);
        result.put(BetCommandType.ERROR, PlayerState.FOLD);
        result.put(BetCommandType.TIMEOUT, PlayerState.FOLD);
        return result;
    }
    public static PlayerState convert(BetCommandType betCommand) {
        return PLAYER_STATE_CONVERSOR.get(betCommand);
    }
}
```

Ejemplo 3.1b

Vemos que hay definidas varias constantes, como el número mínimo de jugadores, el número de cartas por jugador, el máximo número de jugadores o la cantidad de cartas comunitarias. También están definidos como enumerados el tipo de apuesta que puede enviar un jugador, el estado del juego y el estado del jugador. Este último tiene un atributo de tipo *boolean*, que permitirá indicar si en ese estado el jugador va a poder seguir participando en la partida o no. Y finalmente un método estático que permite realiza conversiones entre *BetCommandType* y *PlayerState*.

En este punto vamos a modelar la configuración de la partida, lo haremos en la clase *Settings*, en el ejemplo 3.2a y 3.2b.

```
public class Settings {
    private int maxPlayers;
    private long time;
    private int maxErrors;
    private int maxRounds;
    private long playerChip;
    private long smallBlind;
    private int rounds4IncrementBlind;

    public Settings() {
    }
    public Settings(Settings s) {
        this.maxPlayers = s.maxPlayers;
        this.time = s.time;
        this.maxErrors = s.maxErrors;
        this.playerChip = s.playerChip;
        this.smallBlind = s.smallBlind;
        this.rounds4IncrementBlind = s.rounds4IncrementBlind;
    }
    ...
```

Ejemplo 3.2a

```
    ...
    public int getMaxErrors() {
        return maxErrors;
    }

    public void setMaxErrors(int maxErrors) {
        this.maxErrors = maxErrors;
    }

    public int getMaxPlayers() {
        return maxPlayers;
    }

    public void setMaxPlayers(int maxPlayers) {
        this.maxPlayers = maxPlayers;
    }

    public long getTime() {
        return time;
    }

    public void setTime(long time) {
        this.time = time;
    }

    public long getPlayerChip() {
        return playerChip;
    }

    public void setPlayerChip(long playerChip) {
        this.playerChip = playerChip;
    }
    public long getSmallBlind() {
        return smallBlind;
    }
    public long getBigBlind() {
        return smallBlind * 2;
    }
    public void setSmallBlind(long smallBlind) {
        this.smallBlind = smallBlind;
    }
    public int getMaxRounds() {
        return maxRounds;
    }
    public void setMaxRounds(int maxRounds) {
        this.maxRounds = maxRounds;
    }
    public int getRounds4IncrementBlind() {
        return rounds4IncrementBlind;
    }
    public void setRounds4IncrementBlind(int rounds4IncrementBlind) {
        this.rounds4IncrementBlind = rounds4IncrementBlind;
    }
}
```

Ejemplo 3.2b

El modelado de la información de estado de cada jugador, lo representaremos en la clase *PlayerInfo* en el ejemplo 3.3a y 3.3b.

```
public class PlayerInfo {

    private String name;
    private long chips;
    private long bet;
    private Card[] cards = new Card[TexasHoldEmUtil.PLAYER_CARDS];
    private PlayerState state;
    private int errors;

    public PlayerInfo() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getChips() {
        return chips;
    }

    public boolean isActive(){
        return state.isActive();
    }

    public void setChips(long chips) {
        this.chips = chips;
    }

    public void addChips(long chips) {
        this.chips += chips;
    }

    public long getBet() {
        return bet;
    }

    public void setBet(long bet) {
        this.bet = bet;
    }

    public Card[] getCards() {
        return new Card[]{cards[0], cards[1]};
    }
     ...
```

Ejemplo 3.3a

```
    ...
    public void setCards(Card[] cards) {
        this.cards[0] = cards[0];
        this.cards[1] = cards[1];
    }

    public PlayerState getState() {
        return state;
    }

    public void setState(PlayerState state) {
        this.state = state;
    }

    public int getErrors() {
        return errors;
    }

    public void setErrors(int errors) {
        this.errors = errors;
    }

    public Card getCard(int index) {
        return cards[index];
    }

    public void setCards(Card card0, Card card1) {
        this.cards[0] = card0;
        this.cards[1] = card1;
    }
}
```

Ejemplo 3.3b

La información del juego queda modelada con la clase *GameInfo* en el ejemplo 3.4a, 3.4b y 3.4c.

```
public class GameInfo<P extends PlayerInfo> {

    private int round;
    private int dealer;
    private int playerTurn;
    private TexasHoldEmUtil.GameState gameState;
    private final List<Card> communityCards = new ArrayList<>(COMMUNITY_CARDS);
    private List<P> players;
    private Settings settings;

    public Settings getSettings() {
        return settings;
    }
```

Ejemplo 3.4a

```
...
public void setSettings(Settings settings) {
    this.settings = settings;
}

public int getRound() {
    return round;
}
public void setRound(int round) {
    this.round = round;
}

public int getDealer() {
    return dealer;
}
public void setDealer(int dealer) {
    this.dealer = dealer;
}

public int getPlayerTurn() {
    return playerTurn;
}
public void setPlayerTurn(int playerTurn) {
    this.playerTurn = playerTurn;
}

public TexasHoldEmUtil.GameState getGameState() {
    return gameState;
}
public void setGameState(TexasHoldEmUtil.GameState gameState) {
    this.gameState = gameState;
}

public List<Card> getCommunityCards() {
    return new ArrayList<>(communityCards);
}
public void setCommunityCards(List<Card> communityCards) {
    this.communityCards.clear();
    this.communityCards.addAll(communityCards);
}

public List<P> getPlayers() {
    return new ArrayList<>(players);
}
public void setPlayers(List<P> players) {
    this.players = new ArrayList<>(players);
}
public P getPlayer(int index) {
    return players.get(index);
}

public int getNumPlayers() {
    return players.size();
}
...
```

Ejemplo 3.4b

```java
    ...
    public boolean addPlayer(P player) {
        return this.players.add(player);
    }
    public boolean removePlayer(P player) {
        return this.players.remove(player);
    }

    public boolean addCommunityCard(Card card) {
        boolean result = false;
        if (communityCards.size() < TexasHoldEmUtil.COMMUNITY_CARDS){
            result = communityCards.add(card);
        }
        return result;
    }
    public void clearCommunityCard() {
        this.communityCards.clear();
    }
}
```

Ejemplo 3.4c

La clase *GameInfo*, tiene un parámetro genérico "P", este parámetro sólo puede ser o del tipo *PlayerInfo* o una subclase de él. Además de contar con los getters y setter oportunos, también posee métodos para agregar elementos a la listas o para vaciar la lista, como las de cartas comunitarias o la de jugadores.

Ya en el capítulo 1 vimos cómo utilizar colecciones parametrizadas, las clases genéricas nos brindan la posibilidad de reutilizar código y permiten definir cualquier clase o interfaz como parámetro, salvo tipos primitivos, sin embargo sí se pueden definir Arrays de tipos primitivos como parámetros, además es posible agregar restricciones como estamos haciendo con la clase *GameInfo*, donde el parámetro debe de ser una subclase de *PlayerInfo*.

En otras situaciones interesará definir que el parámetro tenga una restricción inversa, es decir, que sea superclase de otro, para ver un ejemplo real basta con ir a la clase *java.util.Collections*, ahí nos encontramos con métodos estáticos tales cómo sort(List<T> list, Comparator<? Super T> c), donde el segundo parámetro es un comparador definido por un comodín (wild card) que es una super-clase de T pero que en realidad no nos importa que clase es. Dicho de otro modo, para ordenar una lista de elementos necesitamos un comparador, este comparador debe de ser capaz de comparar dos objetos de la clase T o de cualquier superclase de T.

Como ejemplo tenemos una lista de enteros *List<Integer>* y un comparador de números que implementa *Comparator<Number>* de modo que se trata de un comparador válido para el algoritmo por lo que el método *sort* de *Collections* lo puede admitir como válido, pero hay que ser cuidadoso con la definición del método para que el compilador no de un error, en este caso concreto no pondrá ningún problema. En cualquier caso, se trata de una cuestión con gran trasfondo, por lo que conviene leer con detenimiento el libro ***"Java Generics and Collections"***.

Un jugador puede enviar apuestas, estas apuestas las vamos a modelar como comandos de tipo apuesta en la clase *BetCommand*, en el ejemplo 3.5 está el código.

```
public class BetCommand {

    private final BetCommandType type;
    private long chips;

    public BetCommand(BetCommandType type, long chips) {
        ExceptionUtil.checkNullArgument(type, "type");
        ExceptionUtil.checkMinValueArgument(chips, 0L, "chips");
        this.type = type;
        this.chips = chips;
    }

    public BetCommand(BetCommandType type) {
        this(type, 0);
    }

    public BetCommandType getType() {
        return type;
    }

    public long getChips() {
        return chips;
    }

    public void setChips(long chips) {
        this.chips = chips;
    }
}
```

Ejemplo 3.5

La interfaz que debería implementar una clase de tipo jugador será IStrategy, queda ilustrada en el ejemplo 3.6.

```
public interface IStrategy {

    public String getName();

    public BetCommand getCommand(GameInfo<PlayerInfo> state);

    public default void updateState(GameInfo<PlayerInfo> state){}

    public default void check(List<Card> communityCards){}

    public default void onPlayerCommand(String player, BetCommand betCommand){}
}
```

Ejemplo 3.6

Hemos agregado métodos con implementaciones por defecto, es una de las novedades de Java 8, por eso aparece la palabra reservada *default*. De modo que no es obligatorio implementarlos.

La interfaz del controlador general del juego la podemos ver en el ejemplo 3.7.

```
public interface IGameController {

    public void setSettings(Settings settings);

    public boolean addStrategy(IStrategy strategy);

    public void start();

    public void waitFinish();
}
```

Ejemplo 3.7

En el capítulo anterior desarrollamos la implementación del algoritmo de evaluación y la del algoritmo de combinaciones, en para terminar este capítulo vamos a crear una nueva funcionalidad que una ambas, se trata de un algoritmo de evaluación de siete cartas, lo que debe hacer es calcular la mejor combinación de cinco cartas posibles del conjunto de siete. Llamaremos a esta nueva clase *Hand7Evaluator*. En los ejemplos 3.8a y 3.8b vemos como queda.

```
public class Hand7Evaluator {

    public static final int TOTAL_CARDS = PLAYER_CARDS + COMMUNITY_CARDS;
    private final int[] combinatorialBuffer = new int[COMMUNITY_CARDS];
    private final Combination combinatorial =
                                  new Combination(COMMUNITY_CARDS, TOTAL_CARDS);
    private final IHandEvaluator evaluator;
    private final Card[] evalBuffer = new Card[COMMUNITY_CARDS];
    private final Card[] cards = new Card[TOTAL_CARDS];
    private int communityCardsValue = 0;

    public Hand7Evaluator(IHandEvaluator evaluator) {
        this.evaluator = evaluator;
    }

    public void setCommunityCards(List<Card> cc) {
        int i = 0;
        for (Card card : cc) {
            evalBuffer[i] = card;
            cards[i++] = card;
        }
        communityCardsValue = evaluator.eval(evalBuffer);
    }
    ...
}
```

Ejemplo 3.8a

```
    ...
    public int eval(Card c0, Card c1) {
        cards[COMMUNITY_CARDS] = c0;
        cards[COMMUNITY_CARDS + 1] = c1;
        return evalCards();
    }

    static Card[] copy(Card[] src, Card[] target, int[] positions) {
        int i = 0;
        for (int p : positions) {
            target[i++] = src[p];
        }
        return target;
    }

    private int evalCards() {
        combinatorial.clear();
        combinatorial.next(combinatorialBuffer);
        int result = communityCardsValue;
        while (combinatorial.hasNext()) {
            result = Math.max(result, evaluator.eval(
                     copy(cards, evalBuffer, combinatorial.next(combinatorialBuffer))));
        }
        return result;
    }
}
```

Ejemplo 3.8b

Aquí damos por concluido el capítulo 3 y por terminada la capa de la API del juego, para resumir hemos definido dos paquetes: *org.poker.api.core* y *org.poker.api.game*, en el paquete *core* tenemos *Card*, *Deck*, *Hands*, *IHandEvaluator y HandEvaluator*. Son elementos básicos del juego. En el paquete *game* tenemos *TexasHoldEmUtil*, *Settings*, *PlayerInfo*, *GameInfo*, *BetCommand*, *IStrategy*, *IGameController* y *Hand7Evaluator* que son elementos más relacionados con la modalidad concreta del Póquer que vamos a implementar.

Capítulo 4: Máquina de estados

En este punto vamos a crear una pequeña máquina de estados con una funcionalidad mínima, el objetivo es crear una pequeña funcionalidad que permita modelar una parte del juego basándonos en estados y transiciones. Este módulo quedará enmarcado en la estructura global dentro del módulo de “State Machine” (ver diagrama 4.1) y se corresponderá con la paquetería *org.util.statemachine*.

El módulo estará compuesto por una clase que represente el modelo de máquinas de estados, sus propios estados, transiciones, chequeador de transiciones, otra clase que represente una instancia de la máquina de estados y algunas utilidades para la ejecución de estados; estas clases van a tener un parámetro genérico, el contexto que contenga el modelo de datos que se necesita compartir para que todos los elementos funcionen en sintonía sobre una base común.

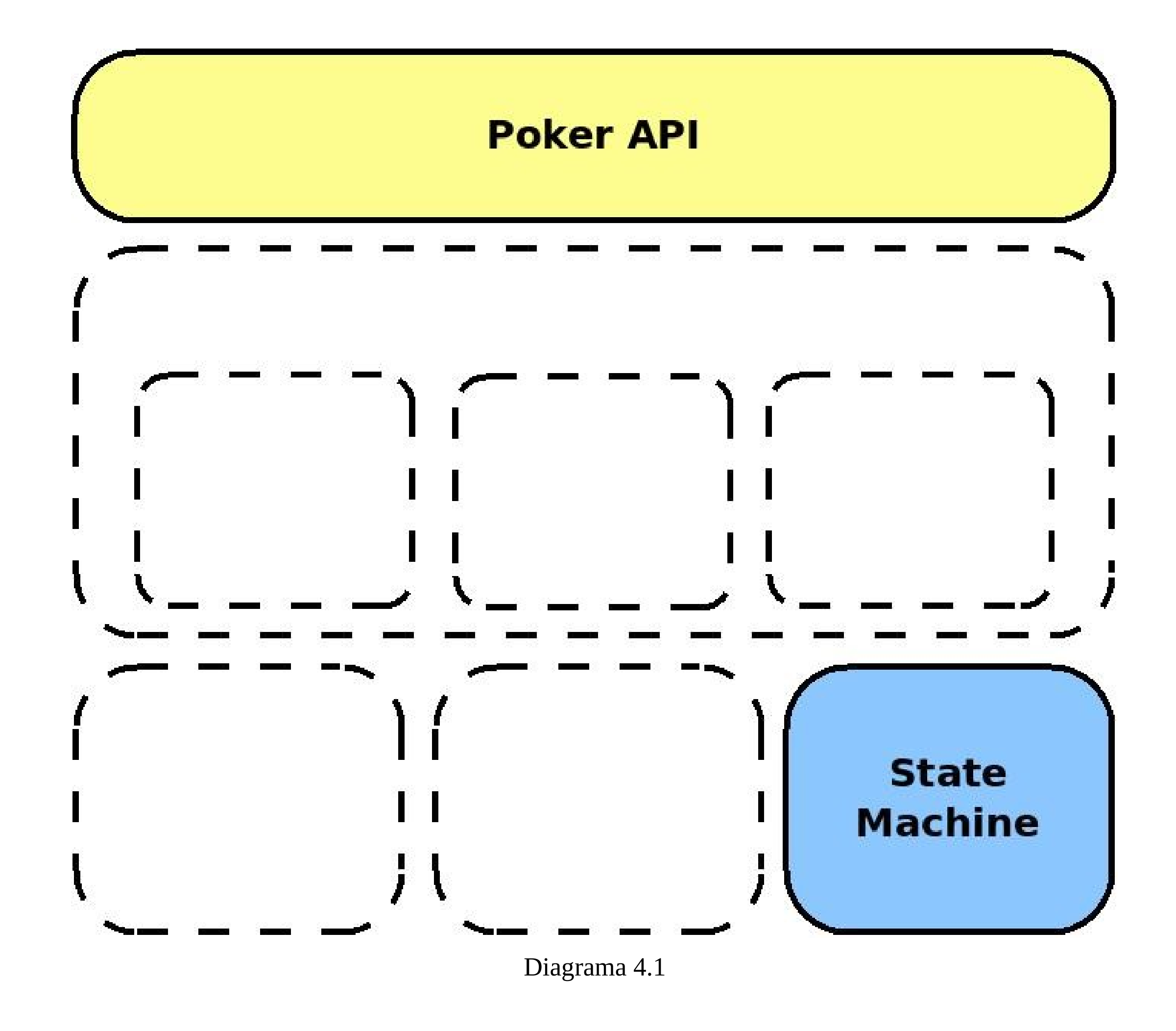


Diagrama 4.1

Por comenzar por la parte más sencilla, vamos a definir un estado como una clase que implemente la interfaz *IState*, con sólo dos métodos: uno para obtener el nombre del estado y otro para "ejecutar" la lógica asociada a él. Este último devuelve un valor *boolean*, que determina si la lógica del estado se ha ejecutado completamente o si se ha quedado detenido en el estado a la espera de que algo lo vuelva a iniciar. En el ejemplo 4.1 tenemos el código de *IState*.

```
public interface IState<T> {

    public String getName();

    public boolean execute(T context);
}
```

Ejemplo 4.1

La siguiente interfaz a definir es la que se debe implementar para determinar que transición será la siguiente a seguir para alcanzar el próximo estado. Si bien es cierto que esta interfaz podría haberla incluido dentro de la clase transición y definir esta como una clase abstracta he optado por la opción de dejarla a un lado como una interfaz de un único método con un objetivo simple, que las implementaciones de la misma sean expresiones lambda, una de las novedades de Java 8.

Esta interfaz la vamos a denominar *IChecker*, en el ejemplo 4.2 vemos el resultado.

```
@FunctionalInterface
public interface IChecker<T> {

    public boolean check(T context);
}
```

Ejemplo 4.2

Para la definición de las transiciones vemos cómo se van a relacionar con las interfaces que acabamos de describir. Una transición va a almacenar la referencia de los estados origen y fin así como la clase que permite realizar la comprobación que determinará si esa es la transición elegida. En el diagrama 4.2 vemos la relación.

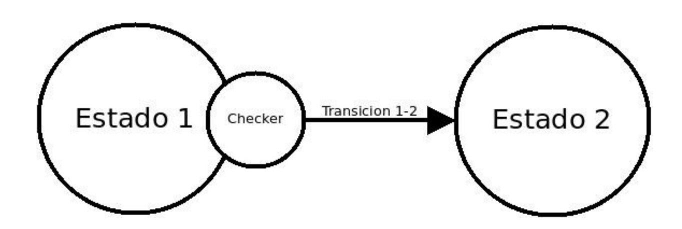


Diagrama 4.2

A continuación vemos en el ejemplo 4.3 la clase *Transition*.

```
public class Transition<T> {

    private final IState<T> origin;
    private final IState<T> target;
    private final IChecker<T> checker;

    public Transition(IState<T> origin, IState<T> target, IChecker<T> checker) {
        this.origin = origin;
        this.target = target;
        this.checker = checker;
    }

    public IState<T> getOrigin() {
        return origin;
    }

    public IState<T> getTarget() {
        return target;
    }

    public IChecker<T> getChecker() {
        return checker;
    }
}
```

Ejemplo 4.3

Un estado podrá tener tantas transiciones como sean necesarias, incluso transiciones a sí mismo, también podrá tener una transición por defecto, aunque está no necesitará un objeto explícito. Los objetos de tipo IC*hecker* no tienen información acerca de estados o transiciones, se limitan a evaluar el contexto para determinar si se cumple o no la condición que determina que se realice ese salto.

Los objetos de tipo *IState* no tienen información sobre sus transiciones ni necesita realizar evaluaciones, su objetivo es ejecutar la lógica asociada a ese estado y devolver un valor *boolean* que determina si el estado ha concluido su ejecución o si por el contrario el estado se encuentra en un modo pausa, esperando que un evento cambie el contexto y dispare nuevamente la ejecución de la lógica del estado.

Cuando un estado ejecuta adecuadamente su lógica y devuelve el valor oportuno, habrá que evaluar cuál de sus transiciones (si es que posee alguna) es la que debe determinar el siguiente estado, si no hubiese ninguna transición, se utilizaría la transición por defecto de este estado, si el siguiente estado fuese nulo, se daría por concluida la ejecución de esa instancia de la máquina de estados.

La Clase *StateMachine* va a modelar la máquina de estados, podemos ver cómo queda en el ejemplo 4.4.

```
public class StateMachine<T> {

    private IState<T> initState = null;
    private final Map<String, IState<T>> defaultTransition = new HashMap<>();
    private final Map<String, List<Transition<T>>> transitions = new HashMap<>();

    public StateMachine() {
    }

    List<Transition<T>> getTransitionsByOrigin(IState<T> state) {
        List<Transition<T>> result = transitions.get(state.getName());
        if (result == null) {
            result = Collections.emptyList();
        }
        return result;
    }

    public void setInitState(IState<T> initState) {
        this.initState = initState;
    }

    public IState<T> getDefaultTransition(IState<T> origin) {
        return defaultTransition.get(origin.getName());
    }

    public void setDefaultTransition(IState<T> origin, IState<T> target) {
        this.defaultTransition.put(origin.getName(), target);
    }

    public void addTransition(Transition<T> transition) {
        IState<T> origin = transition.getOrigin();
        List<Transition<T>> listTransitions = transitions.get(origin.getName());
        if (listTransitions == null) {
            listTransitions = new ArrayList<>();
            transitions.put(origin.getName(), listTransitions);
        }
        listTransitions.add(transition);
    }

    public void addTransition(IState<T> origin, IState<T> target, IChecker<T> checker) {
        addTransition(new Transition<>(origin, target, checker));
    }

    public StateMachineInstance<T> startInstance(T data) {
        return new StateMachineInstance(data, this, initState).execute();
    }
}
```

Ejemplo 4.4

El método *startInstance()* crea una instancia para la ejecución de esta máquina de estados, está clase la vamos a describir más adelante.

La clase *StateMachineInstance* la vemos en el ejemplo 4.5.

```
public class StateMachineInstance<T> {

    private final T context;
    private final StateMachine<T> parent;
    private IState<T> state;
    private boolean finish;
    private boolean pause;

    public StateMachineInstance(T context, StateMachine<T> parent, IState<T> state{
        this.context = context;
        this.parent = parent;
        this.state = state;
        this.finish = false;
    }

    public boolean isFinish() {
        return finish;
    }

    public StateMachineInstance<T> execute() {
        this.pause = false;
        while (state != null && !pause) {
            state = executeState();
        }
        finish = state == null;
        return this;
    }

    public T getContext() {
        return context;
    }

    private IState<T> executeState() {
        pause = !state.execute(context);
        IState<T> result = state;
        if (!pause) {
            for (Transition<T> transition : parent.getTransitionsByOrigin(state)) {
                if (transition.getChecker().check(context)) {
                    return transition.getTarget();
                }
            }
            result = parent.getDefaultTransition(state);
        }
        return result;
    }
}
```

Ejemplo 4.5

Supongamos que tenemos una máquina de estados, con cuatro estados, donde el estado inicial es el “Estado 1”, este tiene dos transiciones definidas y una transición por defecto al “Estado 4” si no se satisface ninguna de las anteriores. El orden de las dos transiciones es importante ya que la evaluación se realizará de forma individual y cuando se pase por una comprobación satisfactoria, se saltará al estado destino definido en esa transición, aunque pudieran haber otras transiciones que también tuviesen una comprobación satisfactoria se ignorarían y se seleccionaría la primera.

Vamos a ilustrarlo con un ejemplo concreto, este tipo de máquina de estados lo podríamos ver en el diagrama 4.3, supongamos que el contexto es un número entero, ningún estado tiene lógica asociada y las transiciones vienen definidas del siguiente modo, al estado 2 si se trata de un número divisible por dos y al estado 3 si el número es divisible por 3, en cualquier otro caso se debe ir al estado 4, que estará marcado como por defecto. Como estos tres estados tienen una transición a un estado nulo (de hecho no tienen ninguna transición definida) se consideran estados finales.

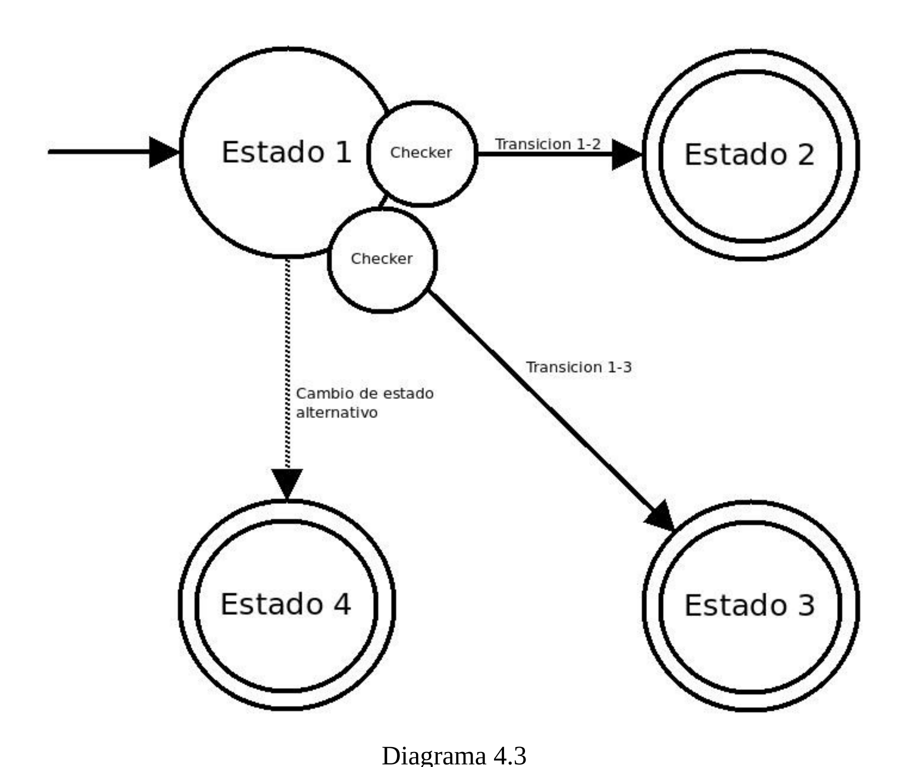


Diagrama 4.3

Vamos a colocar mensajes en la clase *StateMachineInstance* para ver su funcionamiento interno, antes de la ejecución de la lógica de cada estado mostraremos un mensaje con el estado y “Executing...” después de esta ejecución y en función del resultado, mostraremos un mensaje con “[executed]” o “[paused]” y cuando la instancia de la máquina de estados haya terminado su ejecución, mostraremos el mensaje "execute finish".

En el ejemplo 4.6 vemos la implementación en código de la máquina de estados descrita y el resultado de la ejecución para el valor numérico '6'.

```java
public class IntegerStateMachineExample {

    public static void main(String[] args) {
        StateMachine<Integer> sm = new StateMachine<>();
        IntState state1 = new IntState("State 1");
        IntState state2 = new IntState("State 2");
        IntState state3 = new IntState("State 3");
        IntState state4 = new IntState("State 4");
        sm.setInitState(state1);
        sm.addTransition(state1, state2, (n) -> (n % 2) == 0);
        sm.addTransition(state1, state3, (n) -> (n % 3) == 0);
        sm.setDefaultTransition(state1, state4);

        StateMachineInstance<Integer> smi = sm.startInstance(6);
    }

    private static class IntState implements IState<Integer> {

        private String name;

        public IntState(String name) {
            this.name = name;
        }

        @Override
        public String getName() {
            return name;
        }

        @Override
        public boolean execute(Integer context) {
            return true;
        }
    }
}
```

```
22:00:00.000 [main] DEBUG o.u.s.StateMachineInstance - state "State 1" executing...
22:00:00.001 [main] DEBUG o.u.s.StateMachineInstance - state "State 1" [executed]
22:00:00.002 [main] DEBUG o.u.s.StateMachineInstance - state "State 2" executing...
22:00:00.003 [main] DEBUG o.u.s.StateMachineInstance - state "State 2" [executed]
22:00:00.004 [main] DEBUG o.u.s.StateMachineInstance - execute finish
```

Ejemplo 4.6

Podemos comprobar como al definir la transición del estado 1 al 2 antes que la del Estado 1 al 3, esta se ejecutó, y la del Estado 1 al 3 se ignoró. También vemos como las transiciones las hemos podido definir como expresiones lambda gracias a las consideraciones previas.

La máquina de estados es bastante sencilla y potente, pero vamos a incluir una funcionalidad extra: la posibilidad de ejecutar código antes y después de cada estado, con el patrón de diseño decorador.

El decorador simula ser un estado, pero necesita contener un estado, para delegar la ejecución cuando sea preciso, puede ejecutar código antes y después de la lógica del estado anidado, así como saber si el resultado de la ejecución de la lógica del estado, si ha concluido o está en pausa.

No confundir el patrón decorador con el paradigma de la programación orientada a aspectos (AOP), aunque existe alguna similitud, las diferencias son enormes.

En el diagrama 4.4 el aspecto está representado en color naranja.

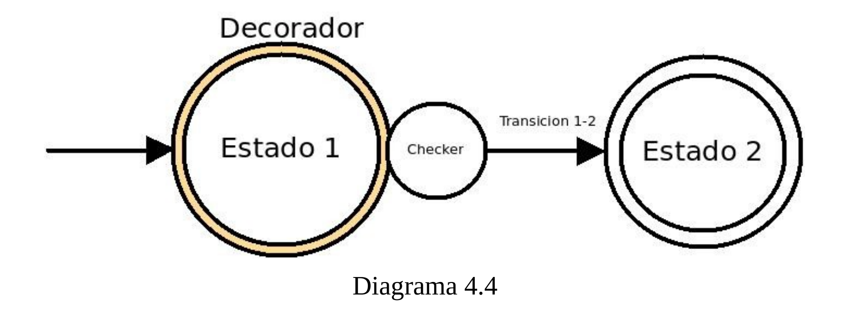


Diagrama 4.4

En el ejemplo 4.7 vemos un decorador que ejecutará un *Runnable* después de que el estado finalice su ejecución. En caso de pausa no se realizará esta ejecución.

```
public class AfterStateDecorator<T> implements IState<T> {

    private final IState<T> state;
    private final Runnable listener;

    public AfterStateDecorator(IState<T> state, Runnable listener) {
        this.state = state;
        this.listener = listener;
    }

    @Override
    public String getName() {
        return state.getName();
    }

    @Override
    public boolean execute(T context) {
        boolean result = state.execute(context);
        if (result) {
            listener.run();
        }
        return result;
    }
}
```

Ejemplo 4.7

Y en el Ejemplo 4.8 está la implementación de un decorador que ejecutará un *Runnable* antes de que el estado inicie su ejecución, ignorando el reinicio de la ejecución del estado anidado debido a un caso de pausa.

```
public class BeforeStateDecorator<T> implements IState<T> {

    private final IState<T> state;
    private final Runnable listener;
    private boolean executed = true;

    public BeforeStateDecorator(IState<T> state, Runnable listener) {
        this.state = state;
        this.listener = listener;
    }

    @Override
    public String getName() {
        return state.getName();
    }

    @Override
    public boolean execute(T context) {
        if (executed) {
            listener.run();
        }
        executed = state.execute(context);
        return executed;
    }
}
```

Ejemplo 4.8

Para construir de un modo más compacto la creación de estos decoradores, vamos a utilizar el patrón de diseño builder. Ya hemos hablado de patrones anteriormente, sobre esta cuestión se ha escrito mucha literatura y seguramente más de un académico hubiera preferido hablar de patrones de diseño antes que de TDD y uno de los motivos de más peso tiene en mi opinión es el hecho de que el patrón de diseño DI (Inyección de dependencias) y la IoC (Inversión de control) son un pilar muy importante para elevar el TDD a su máxima expresión. Lo cierto es que he pretendido que el uso de estas estrategias surgiese de una necesidad concreta y desarrollarlas de un modo natural.

Y en el ejemplo 4.9 vemos la implementación de nuestro patrón de diseño builder.

```
public class StateDecoratorBuilder<T> {

    private IState<T> state;

    private StateDecoratorBuilder(IState<T> state) {
        this.state = state;
    }

    public static <T> StateDecoratorBuilder<T> create(IState<T> state) {
        return new StateDecoratorBuilder<>(state);
    }

    public StateDecoratorBuilder<T> after(Runnable r) {
        this.state = new AfterStateDecorator<>(state, r);
        return this;
    }

    public StateDecoratorBuilder<T> before(Runnable r) {
        this.state = new BeforeStateDecorator<>(state, r);
        return this;
    }

    public IState<T> build() {
        return state;
    }

    public static <T> IState<T> after(IState<T> state, Runnable r) {
        return new AfterStateDecorator<>(state, r);
    }

    public static <T> IState<T> before(IState<T> state, Runnable r) {
        return new BeforeStateDecorator<>(state, r);
    }
}
```

Ejemplo 4.9

Vemos como además de la implementación del patrón builder hemos agregado dos métodos para encapsular la creación de los decoradores anteriormente definidos.

Aquí damos por concluido el módulo de la máquina de estados.

Capítulo 5: Eventos y concurrencia

A lo largo de este capítulo vamos a ver la creación del módulo *distpacher*, lo vamos a encuadrar dentro de la paquetería *org.util.dispatccher.*

El módulo *dispatcher* se compone por dos clases y por dos interfaces, una de las clases es el evento que utilizaremos en el juego, lo llamaremos *GameEvent*; una de las interfaces se llamará *IGameEventDispatcher* y se encargará de definir la interfaz que deberá tener un despachador de eventos. Luego está la clase *GameEventDispatcher*, que implementa la interfaz anterior y finalmente estará la interfaz *IGameEventProcessor*, que va a resultar muy útil en el futuro. Como se puede apreciar, se trata de un patrón de diseño en toda regla: Listener.

En el diagrama 5.1 vemos donde va a encajar este nuevo módulo.

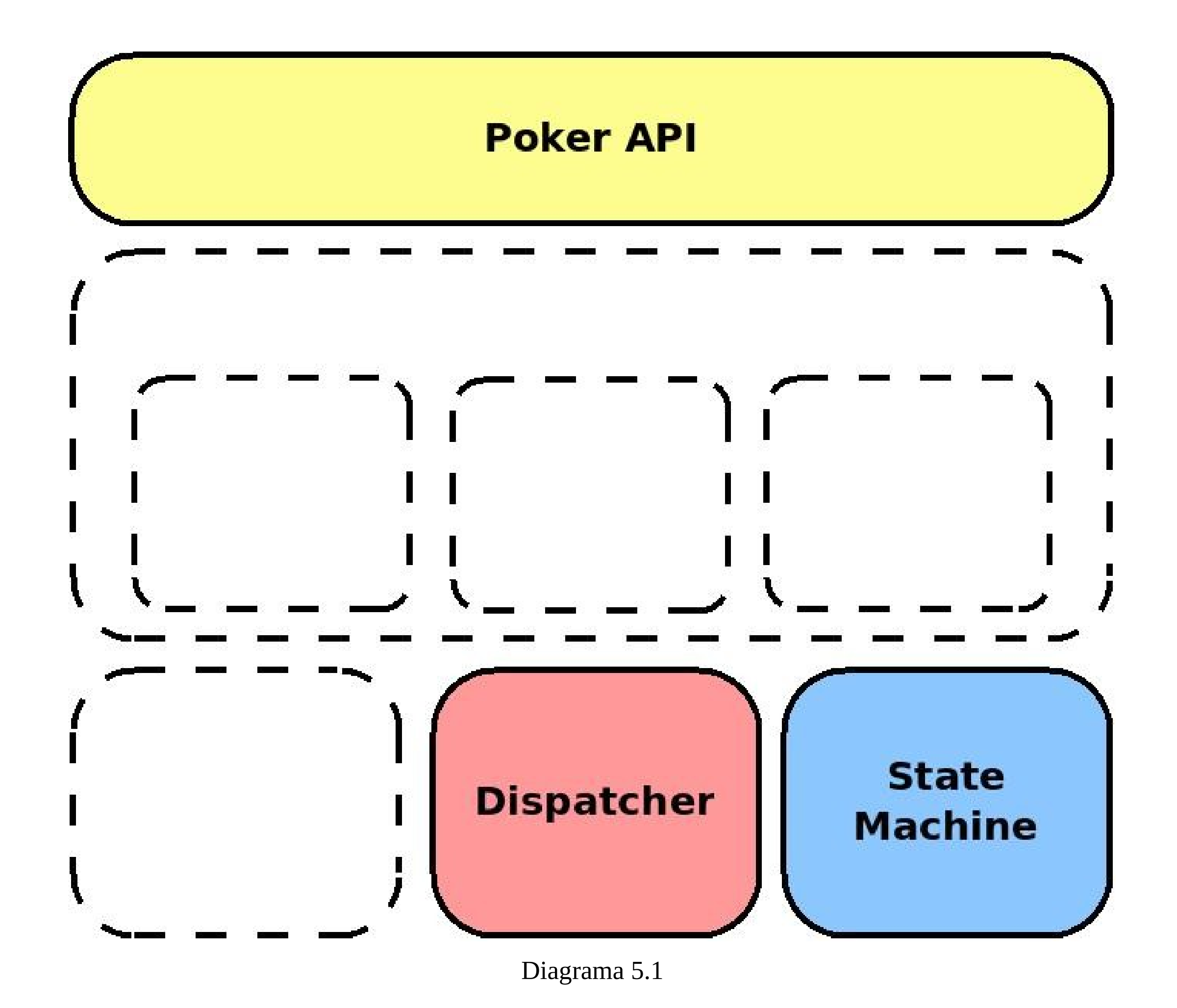


Diagrama 5.1

En el ejemplo 5.1 vemos como queda la clase *GameEvent*. No es más que un objeto plano con getters y setters que sirve para transmitir información de los eventos producidos. Como el contenido del mensaje (atributo *payload*) no puedo concretarlo más he optado por declararlo de tipo *Object*, no es una estrategia que me guste pero es un modo sencillo de conseguir cubrir la necesidad de versatilidad de forma sencilla.

```
public class GameEvent {

    private String type;
    private String source;
    private Object payload;

    public GameEvent() {
    }

    public GameEvent(String type, String source) {
        this.source = source;
        this.type = type;
    }

    public GameEvent(String type, String source, Object payload) {
        this.source = source;
        this.type = type;
        this.payload = payload;
    }

    public String getSource() {
        return source;
    }

    public void setSource(String source) {
        this.source = source;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public Object getPayload() {
        return payload;
    }

    public void setPayload(Object payload) {
        this.payload = payload;
    }
}
```

Ejemplo 5.1

La interfaz *IGameEventDispatcher* la vemos en el ejemplo 5.2.

```
public interface IGameEventDispatcher extends Runnable {

    public void dispatch(GameEvent event);
    public void exit();
}
```

Ejemplo 5.2

La interfaz *IGameEventProcessor* la tenemos en el ejemplo 5.3.

```
@FunctionalInterface
public interface IGameEventProcessor<T> {

    public void process(T target, GameEvent event);
}
```

Ejemplo 5.3

La implementación de la interfaz *IGameEventDispatcher* la desarrollamos en la clase *GameEventDispatcher* y la vemos en el ejemplo 5.4a y 5.4b.

```
public class GameEventDispatcher<T> implements IGameEventDispatcher {
    private static final Logger LOGGER = LoggerFactory.getLogger(….class);

    public static final String EXIT_EVENT_TYPE = "exit";
    private final Map<String, IGameEventProcessor<T>> processors;
    private final T target;
    private List<GameEvent> events = new ArrayList<>();
    private volatile boolean exit = false;
    private ExecutorService executors;

    public GameEventDispatcher(T target, Map<String, IGameEventProcessor<T>> p,
                               ExecutorService executors) {
        this.target = target;
        this.processors = p;
        this.executors = executors;
    }

    public synchronized void dispatch(GameEvent event) {
        events.add(event);
        this.notify();
    }
    ...
```

Ejemplo 5.4a

```
    ...
    @Override
    public synchronized void exit() {
        exit = true;
        this.notify();
    }

    private void doTask() throws InterruptedException {
        List<GameEvent> lastEvents;
        synchronized (this) {
            if (events.isEmpty()) {
                this.wait();
            }
            lastEvents = events;
            events = new ArrayList<>();
        }
        for (int i = 0; i < lastEvents.size() && !exit; i++) {
            GameEvent event = lastEvents.get(i);
            if (EXIT_EVENT_TYPE.equals(event.getType())) {
                exit = true;
            } else {
                process(event);
            }
        }
    }

    @Override
    public void run() {
        while (!exit) {
            try {
                doTask();
            } catch (InterruptedException ex) {
                  LOGGER.error("GameEventDispatcher<" + … ,target, ex);
            }
        }
        executors.shutdown();
    }
}
```

Ejemplo 5.4b

La clase *GameEventDispatcher* se apoya en implementaciones de la interfaz *IGameEventProcessor*, que al ser una interfaz de un sólo método, podemos implementarla con expresiones lambda, siguiendo la estrategia del capítulo anterior. Por este motivo tenemos la interfaz *IgameEventProcessor* anotada con *@FunctionalInterface*.

Aparece por primera vez el modificador de método *synchronized*, este nos ayuda a proteger el código, en estas regiones únicamente habrá un hilo en ejecución, impidiendo que en determinadas condiciones de carrera, la ejecución de varios hilos de lugar errores de concurrencia, estos errores se dan de forma impredecible, no tienen por qué generar una excepción y en muchos casos pueden resultar indetectables.

Esta es la primera vez que hablamos de clases que tienen que gestionar bloques de lógica en entornos multi-hilo, la programación concurrente es bastante compleja ya que el software puede funcionar de forma correcta hasta que en un momento fortuito se comporte de forma inesperada.

Las clases que como esta tienen protección para entornos multi-hilo se denominan *ThreadSafe* (Seguras para hilos), si bien es cierto que en la especificación estándar no se recoge ningún anotación para informar de sobre este cariz, hay algunas sugerencias recogidas en el libro "***Java concurrency in practice***". En mi opinión esta no es una cuestión banal, ya que cuando desarrollamos, tratamos de utilizar clases a modo de constantes en entornos multi-hilo sin ser muy conscientes, lo que puede acabar produciendo dolores de cabeza por errores inesperados, un ejemplo claro es *java.text.SimpleDateFormat*, que es *NotThreadSafe*.

Otra cuestión es la utilización de clases seguras para hilos, cuando generalmente se utilizan en ámbitos protegidos a la concurrencia, existiendo clases alternativas sin esta protección, como ocurre con *StringBuffer* (*ThreadSafe*) y *StringBuilder* (*NotThreadSafe*), tienen los mismos métodos públicos y en ámbitos protegidos para la concurrencia la segunda es la más adecuada. Un ejemplo del ámbito protegido es la declaración de una variable local en un método, y que ninguna referencia a esa variable salga de ese ámbito, ni se almacene en un atributo o colección que pueda quedar expuesta.

Para lograr esta protección multi-hilo hay que ser muy cauteloso utilizando los modificadores *synchronized*, ya que estos puede producir deterioros importantes en el rendimiento de una aplicación, pueden crear cuellos de botella artificiales; en otros casos, tienen un comportamiento más dramático, llegando a producir deadlock, es decir que dos o más hilos necesiten bloquear varios recursos, con tan mala suerte que el que unos necesiten este bloqueado por otros y viceversa, de tal modo que ninguno sea capaz de continuar y queden bloqueados indefinidamente.

Sobre programación concurrente en java hay mucha literatura, requiere una atención por los detalles difícil de dominar, no obstante existen numerosas estrategias y algoritmos sencillos, que aplicándolos adecuadamente se consigue evitar muchos errores comunes en este tipo de entornos, tales como el algoritmo del banquero de Dijsktra, o tratando de evitar acumular varios bloqueos de recursos si no resulta completamente imprescindible.

Cuando el arte de la prevención de deadlock no se ha podido llevar a cabo convenientemente y se sospecha que nuestro software en ejecución puede estar sufriendo algún interbloqueo, tenemos herramientas de diagnóstico y monitorización como jstack, jConsole y jstat que vienen con el Java Development Kit de Oracle. Por este y por otros muchos motivos es muy útil conocer las herramientas que tiene la plataforma Java que se esté utilizando.

Volviendo a la implementación de *GameEventDispatcher*, podemos apreciar las llamadas a dos métodos heredados directamente de la clase *Object*, *notify*() en el método *dispatch(GameEvent)* y *wait*() en el método *doTask*(). Estos métodos tienen una utilidad muy interesante, se pueden bloquear la ejecución de un hilo *wait()* a la espera de que un evento haga que se ejecute un *notify()* y quede desbloqueado el hilo anterior, también es posible que se bloqueen varios hilos y querer liberarlos a todos con la llamada a *notifyAll()*. Todos estos métodos sólo se pueden llamar sobre un objeto que se encuentre en un bloque de código sincronizado por sí mismo.

Estás utilidades pretenden sincronizar la lista de eventos a procesar, de tal modo que cuando el hilo del *dispatcher* no tiene eventos se quedará bloqueado a la espera de un evento, cuando otro hilo agregue un nuevo evento a la lista mediante la invocación del método *dispatch*() se notifica al hilo *dispatcher* para que se desbloquee y pueda procesar los nuevos eventos, hasta que la lista de eventos quede otra vez vacía y se vuelva a bloquear.

El hilo del *dispatcher* será un hilo propio que deberá iniciarse mediante un objeto *Thread* independiente desde fuera de este módulo, internamente utilizará un grupo de hilos para el procesado, este grupo debe llegar ya configurado al constructor; El motivo de esta implementación es evitar que se pueda "secuestrar" el hilo del *dispatcher* durante el procesado de un evento ya que este hecho daría como resultado la acumulación de nuevos eventos sin despachar hasta el final de la ejecución del programa. En este punto damos por concluido el módulo *dispatcher*.

Capítulo 6: Temporizador

Una parte esencial del juego es el temporizador, aunque en el juego real no exista como tal, es una cuestión importante que permite conseguir cierta fluidez durante la ejecución del juego, ya que si un jugador se excede del tiempo marcado en la configuración se le penalizará, retirándole de la ronda, después se incrementará su contador de errores, hasta que llegue al límite de errores permitido y sea descalificado.

En el diagrama 6.1 vemos donde queda el módulo temporizador, la paquetería a emplear será *org.util.timer*. Este módulo va a contener una interfaz *IGameTimer* y una clase que la implementa *GameTimer*.

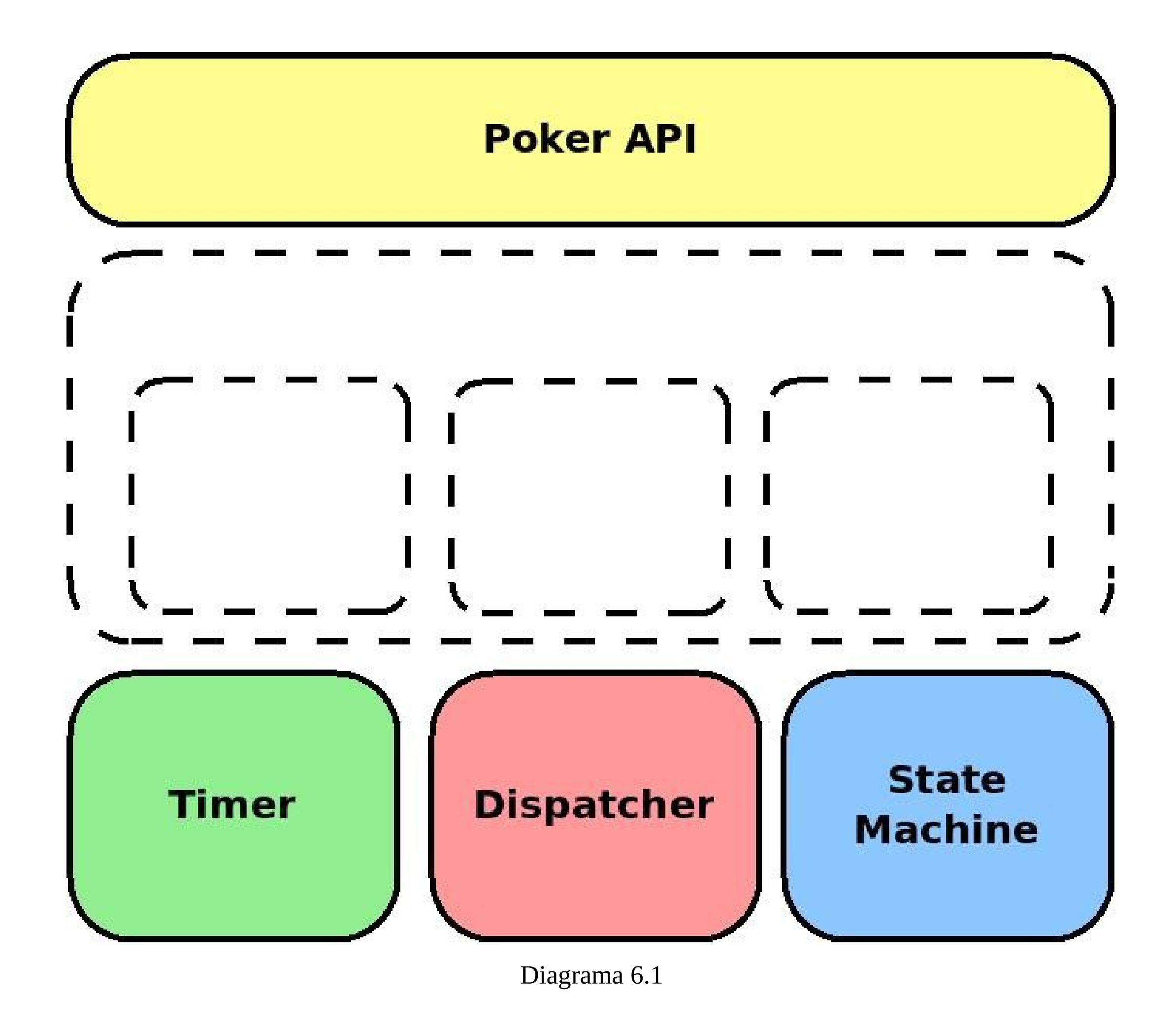


Diagrama 6.1

La interfaz del temporizador va a contener varios métodos para utilizar esta funcionalidad, así como un a referencia a un objeto que implemente la interfaz *IGameEventDispatcher* definida en el capítulo anterior, este objeto resultará de utilidad para notificar eventos de tipo time-out generados por el temporizador. La interfaz del temporizador la podemos ver en el ejemplo 6.1.

```
public interface IGameTimer extends Runnable {

    public void exit();
    public long getTime();
    public void resetTimer(Long timeroutId);
    public void setTime(long time);
    public IGameEventDispatcher getDispatcher();
    public void setDispatcher(IGameEventDispatcher dispatcher);
}
```

Ejemplo 6.1

La implementación de esta interfaz la encontramos en la clase *GameTimer*, y la vemos en los ejemplos 6.2a y 6.2b.

```
public class GameTimer implements IGameTimer {

    private static final Logger LOGGER = LoggerFactory.getLogger(GameTimer.class);
    public static final String TIMEOUT_EVENT_TYPE = "timeOutCommand";

    private final String source;
    private long time;
    private IGameEventDispatcher dispatcher;
    private boolean reset = false;
    private volatile boolean exit = false;
    private final ExecutorService executors;
    private Long timeoutId;


    public GameTimer(String source, ExecutorService executors) {
        this.source = source;
        this.executors = executors;
    }

    @Override
    public synchronized IGameEventDispatcher getDispatcher() {
        return dispatcher;
    }

    @Override
    public synchronized void setDispatcher(IGameEventDispatcher dispatcher) {
        this.dispatcher = dispatcher;
    }
    ...
```

Ejemplo 6.2a

```java
    ...
    @Override
    public long getTime() {
        return time;
    }

    @Override
    public void setTime(long time) {
        this.time = time;
    }

    @Override
    public synchronized void resetTimer(Long timeoutId) {
        this.timeoutId = timeoutId;
        this.reset = true;
        notify();
    }

    @Override
    public synchronized void exit() {
        this.exit = true;
        this.reset = false;
        this.player = null;
        notify();
    }

    @Override
    public void run() {
        LOGGER.debug("run");
        while (!exit) {
            try {
                doTask();
            } catch (InterruptedException ex) {
                LOGGER.error("Timer interrupted", ex);
            }
        }
        LOGGER.debug("finish");
    }

    private synchronized void doTask() throws InterruptedException {
        if (timeoutId == null) {
            wait();
        }
        if ( timeoutId != null) {
            reset = false;
            wait(time);
            if (!reset && timeoutId != null) {
                GameEvent event=new GameEvent(TIMEOUT_EVENT_TYPE, source, timeoutId);
                executors.execute(() -> dispatcher.dispatch(event));
                player = null;
            }
        }
    }
}
```

Ejemplo 6.2b

La implementación de la clase *GameTimer* tiene multitud de similitudes con respecto a la clase *GameEventDispatcher*, como protección para entornos multi-hilo (ThreadSafe), además de utilizar su propio hilo y usar un grupo de hilos para interactuar con terceros (en este caso un objeto de tipo *IGameEventDispatcher).*

Una curiosidad, al definir una clase anónima o una expresión lambda no ha hecho falta declarar la variable local como final, esto contrasta con lo que ocurría en especificaciones anteriores de Java.

Veamos unos ejemplos con código compilable que podríamos haber utilizado en el método privado *doTask* de la clase *GameTimer*:

Java 8 con una expresión lambda: Ejemplo 6.3.

```
GameEvent event = new GameEvent(TIMEOUT_EVENT_TYPE, source, player);
executors.execute(() -> dispatcher.dispatch(event));
```

Ejemplo 6.3

Java 8 con una clase anónima: Ejemplo 6.4.

```
GameEvent event = new GameEvent(TIMEOUT_EVENT_TYPE, source, player);
executors.execute(new Runnable() {
    @Override
    public void run() {
        dispatcher.dispatch(event);
    }
});
```

Ejemplo 6.4

Java 7 con una clase anónima: Ejemplo 6.5.

```
final GameEvent event = new GameEvent(TIMEOUT_EVENT_TYPE, source, player);
executors.execute(new Runnable() {
    @Override
    public void run() {
        dispatcher.dispatch(event);
    }
});
```

Ejemplo 6.5

Aquí concluimos con el capítulo dedicado al módulo temporizador.

Capítulo 7: Modelo de Póquer para la máquina de estados

Este capítulo tratará del módulo que representa el modelo, lo utilizaremos en la máquina de estados para modelar el juego, se trata de definir la información necesaria durante la ejecución de la partida. Este módulo lo vamos a encuadrar dentro del motor del juego que tiene la paquetería *org.poker.engine*, en concreto dentro del sub-paquete *org.poker.engine.model* y va a estar compuesto por tres clases, una para representar datos de cada jugador, otra para el contexto de la máquina de estados y la última como clase de utilidades.

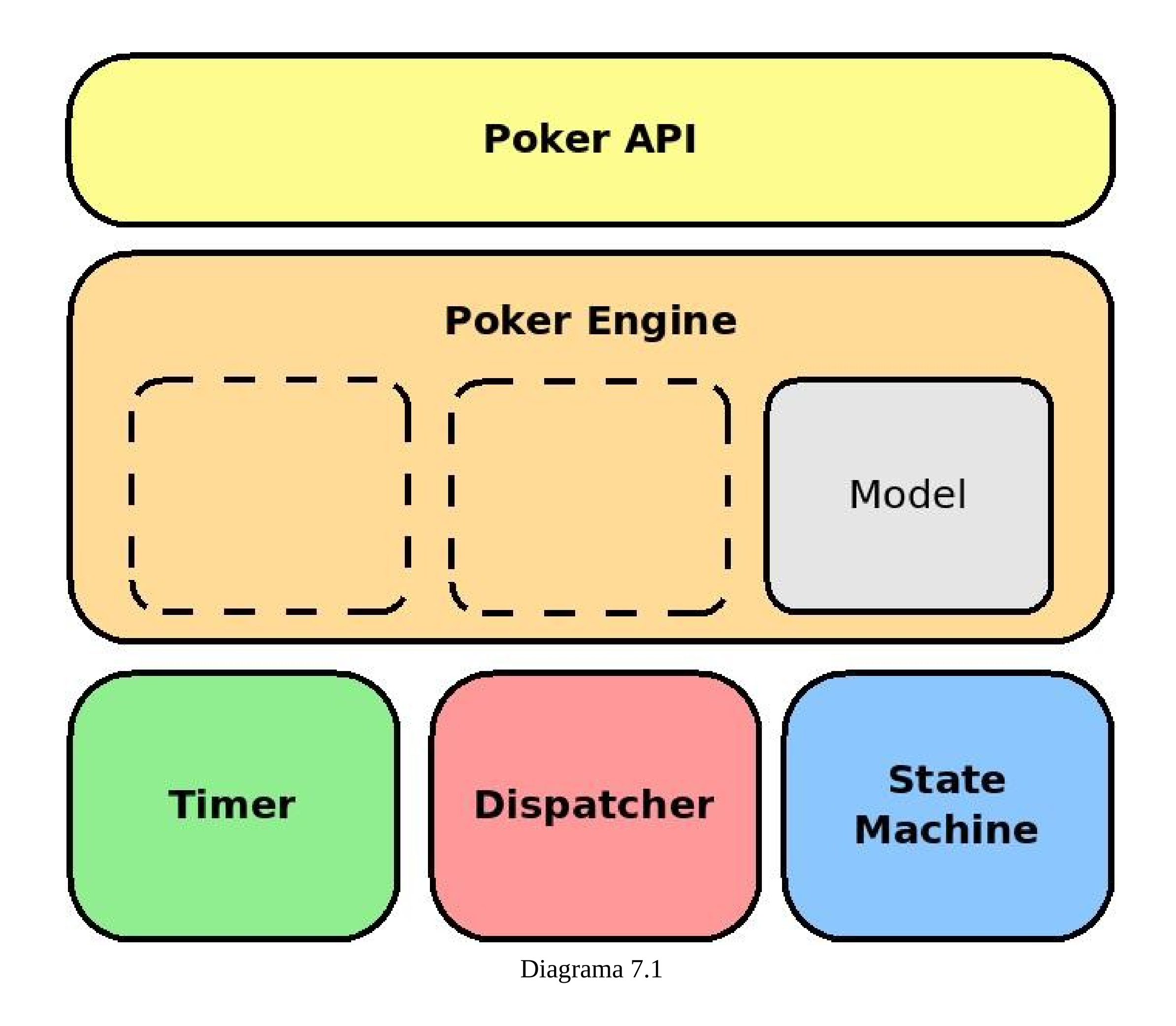


Diagrama 7.1

La primera clase que vamos a definir es la del Jugador dentro de la lógica de la máquina de estados, como comparte muchas cosas en común con la clase *PlayerInfo* del módulo del API, vamos a extender de ella, esta nueva clase la llamaremos *PlayerEntity*, la encontraremos en el ejemplo 7.1.

```
public class PlayerEntity extends PlayerInfo {

    private int handValue = 0;
    private BetCommand betCommand;
    private boolean showCards;

    public PlayerEntity() {
    }

    public boolean showCards() {
        return showCards;
    }

    public void showCards(boolean showCards) {
        this.showCards = showCards;
    }

    public BetCommand getBetCommand() {
        return betCommand;
    }

    public void setBetCommand(BetCommand betCommand) {
        this.betCommand = betCommand;
    }

    public int getHandValue() {
        return handValue;
    }

    public void setHandValue(int handValue) {
        this.handValue = handValue;
    }
}
```

Ejemplo 7.1

En esta clase extendida hemos agregado el valor de la mano (cero si no se ha llegado al estado de evaluar manos), el último comando ejecutado y un indicador para saber si este jugador debe mostrar las cartas o no, el resto viene heredado de la clase padre.

Esta clase va a resultar muy útil para almacenar el contexto global de cada jugador a lo largo de todos los estadios de una ronda, aunque los atributos agregados sean para la ronda de apuestas y la etapa *ShowDown*, conviene recordar que contiene por herencia atributos para saber su estado, el número de errores que ha cometido, el número de fichas o las fichas total que este apostando en la ronda actual además de sus dos cartas individuales.

A continuación vamos a desarrollar la clase *ModelContext*, para guardar la información del juego que se necesitarán a lo largo de la partida. Una clase en la que nos apoyaremos mucho, será la clase *GameInfo* del módulo de la API, ya que esta clase contiene mucha información necesaria. En los ejemplos 7.2a, 7.2b, 7.2c y 7.2d, tenemos el desarrollo de esta clase.

```
public class ModelContext {

    private final GameInfo<PlayerEntity> gameInfo = new GameInfo<>();
    private final Map<String, PlayerEntity> playersByName;
    private int activePlayers;
    private long highBet;
    private Deck deck;
    private int playersAllIn;
    private BetCommand lastBetCommand;
    private PlayerEntity lastPlayerBet;
    private int bets = 0;

    public ModelContext(Settings settings) {
        this.gameInfo.setSettings(settings);
        this.gameInfo.setPlayers(new ArrayList<>(TexasHoldEmUtil.MAX_PLAYERS));
        this.playersByName = new HashMap<>(settings.getMaxPlayers());
        this.highBet = 0;
    }

    public Deck getDeck() {
        return deck;
    }

    public void setDeck(Deck deck) {
        this.deck = deck;
    }

    public int getPlayersAllIn() {
        return playersAllIn;
    }

    public void setPlayersAllIn(int playersAllIn) {
        this.playersAllIn = playersAllIn;
    }

    public int getNumPlayers() {
        return gameInfo.getNumPlayers();
    }

    public boolean addPlayer(String playerName) {
        PlayerEntity player = new PlayerEntity();
        player.setName(playerName);
        player.setChips(gameInfo.getSettings().getPlayerChip());
        this.playersByName.put(playerName, player);
        return this.gameInfo.addPlayer(player);
    }
    ...
```

Ejemplo 7.2a

```
...
public boolean removePlayer(final String playerName) {
    return gameInfo.removePlayer(playersByName.get(playerName));
}

public long getHighBet() {
    return highBet;
}

public void setHighBet(long highBet) {
    this.highBet = highBet;
}

public int getDealer() {
    return gameInfo.getDealer();
}

public void setDealer(int dealer) {
    this.gameInfo.setDealer(dealer);
}

public int getRound() {
    return gameInfo.getRound();
}

public void setRound(int round) {
    this.gameInfo.setRound(round);
}

public void setGameState(GameState gameState) {
    this.gameInfo.setGameState(gameState);
}

public GameState getGameState() {
    return gameInfo.getGameState();
}

public String getPlayerTurnName() {
    String result = null;
    int turnPlayer = gameInfo.getPlayerTurn();
    if (turnPlayer >= 0){
        result = gameInfo.getPlayer(turnPlayer).getName();
    }
    return result;
}

public int getPlayerTurn() {
    return gameInfo.getPlayerTurn();
}

public void setPlayerTurn(int playerTurn) {
    this.gameInfo.setPlayerTurn(playerTurn);
}
...
```

Ejemplo 7.2b

```
...
public Settings getSettings() {
    return gameInfo.getSettings();
}

public List<Card> getCommunityCards() {
    return gameInfo.getCommunityCards();
}

public void setCommunityCards(List<Card> communityCards) {
    gameInfo.setCommunityCards(communityCards);
}

public List<PlayerEntity> getPlayers() {
    return this.gameInfo.getPlayers();
}

public PlayerEntity getPlayer(int player) {
    return this.gameInfo.getPlayer(player);
}

public void setPlayers(List<PlayerEntity> newPlayers) {
    this.gameInfo.setPlayers(newPlayers);
    this.playersByName.clear();
    newPlayers.stream().forEach(p -> this.playersByName.put(p.getName(), p));
}

public PlayerEntity getPlayerByName(String playerName) {
    return playersByName.get(playerName);
}

public int getActivePlayers() {
    return activePlayers;
}

public void setActivePlayers(int activePlayers) {
    this.activePlayers = activePlayers;
}

public void lastResultCommand(PlayerEntity player, BetCommand resultCommand) {
    this.lastPlayerBet = player;
    this.lastBetCommand = resultCommand;
}

public BetCommand getLastBetCommand() {
    return lastBetCommand;
}

public void setLastBetCommand(BetCommand resultLastBetCommand) {
    this.lastBetCommand = resultLastBetCommand;
}
...
```

Ejemplo 7.2c

```
    ...
    public PlayerEntity getLastPlayerBet() {
        return lastPlayerBet;
    }

    public void setLastPlayerBet(PlayerEntity lastPlayerBet) {
        this.lastPlayerBet = lastPlayerBet;
    }

    public int getBets() {
        return bets;
    }

    public void setBets(int bets) {
        this.bets = bets;
    }

    public int addCommunityCards(int numCards) {
        boolean added = true;
        int i = 0;
        while (i < numCards && added) {
            added = gameInfo.addCommunityCard(deck.obtainCard());
            if (added){
                i++;
            }
        }
        return i;
    }

    public void clearCommunityCard() {
        gameInfo.clearCommunityCard();
    }
}
```

Ejemplo 7.2d

En el capítulo 3 definimos la clase *GameInfo* parametrizada, donde el parámetro P se define como una clase derivada de *PlayerInfo* (o la propia clase *PlayerInfo*). Ahora aprovechamos ese diseño para crear un contexto con la información del juego y la clase *PlayerEntity*, la cual nos proporciona la información extra que necesitamos.

La clase *ModelContext* tiene toda la información necesaria para gestionar la situación del juego en cada momento. Esta información será empleada por componentes de una máquina de estados.

Para cerrar el módulo tenemos la clase *ModelUtil* que se trata de una clase de utilidades.

```
public final class ModelUtil {

    public static final int NO_PLAYER_TURN = -1;

    private ModelUtil() {
    }

    public static boolean range(int min, int max, int value) {
        return min <= value && value < max;
    }

    public static int nextPlayer(ModelContext model, int turn) {
        int result = NO_PLAYER_TURN;
        int players = model.getNumPlayers();
        if (players > 1 && range(0, players, turn)) {
            int i = (turn + 1) % players;
            while (i != turn && result == NO_PLAYER_TURN) {
                if (model.getPlayer(i).isActive()) {
                    result = i;
                } else {
                    i = (i + 1) % players;
                }
            }
            result = checkNextPlayer(model, result);
        }
        return result;
    }

    private static int checkNextPlayer(ModelContext model, int index) {
        int result = index;
        if (result != NO_PLAYER_TURN
                && model.getPlayer(result).getBet() == model.getHighBet()
                &&(model.getPlayer(result).getState() != PlayerState.READY
                || model.getActivePlayers() == 1)) {
            result = NO_PLAYER_TURN;
        }
        return result;
    }

    public static void playerBet(ModelContext model, PlayerEntity player,
                                 BetCommandType betCommand, long chips) {
        if (betCommand == BetCommandType.ALL_IN) {
            model.setPlayersAllIn(model.getPlayersAllIn() + 1);
            model.setActivePlayers(model.getActivePlayers() - 1);
        } else if (betCommand == BetCommandType.FOLD ||
                   betCommand == BetCommandType.TIMEOUT) {
            model.setActivePlayers(model.getActivePlayers() - 1);
        }
        playerBet(player, chips);
        model.setHighBet(Math.max(model.getHighBet(), player.getBet()));
        model.setBets(model.getBets() + 1);
    }
    ...
```

Ejemplo 7.3a

```
    ...
    public static void playerBet(PlayerEntity player, long chips) {
        player.setBet(player.getBet() + chips);
        player.setChips(player.getChips() - chips);
    }

    public static void incrementErrors(PlayerEntity player, Settings settings) {
        int errors = player.getErrors() + 1;
        player.setErrors(errors);
        if (errors >= settings.getMaxErrors()) {
            player.setState(PlayerState.OUT);
            player.setChips(0);
        } else {
            player.setState(PlayerState.FOLD);
        }
    }
}
```

Ejemplo 7.3b

Vemos como la clase *ModeloUtil* proporciona funcionalidad para extraer el turno del siguiente jugador en un momento dado de la partida, también permite integrar en el modelo la apuesta de un jugador o incrementar el número de errores verificando si ha llegado al límite en el que debe quedar descalificado.

Aquí concluimos con el capítulo 7, bastante pequeño en número de ficheros pero modela la información que se utilizará en el contexto del juego.

Capítulo 8: Estados del Póquer

En este capítulo vamos a ver el módulo que define los estados del Póquer, habrá etapas del Texas hold'em que se tengan cierta correspondencia con los estados que vamos a definir. Estos estados tienen una lógica asociada y es en este capítulo donde vamos a ver como la implementación de esa lógica modifica la información del contexto y conseguimos tener coherencia con las reglas del juego.

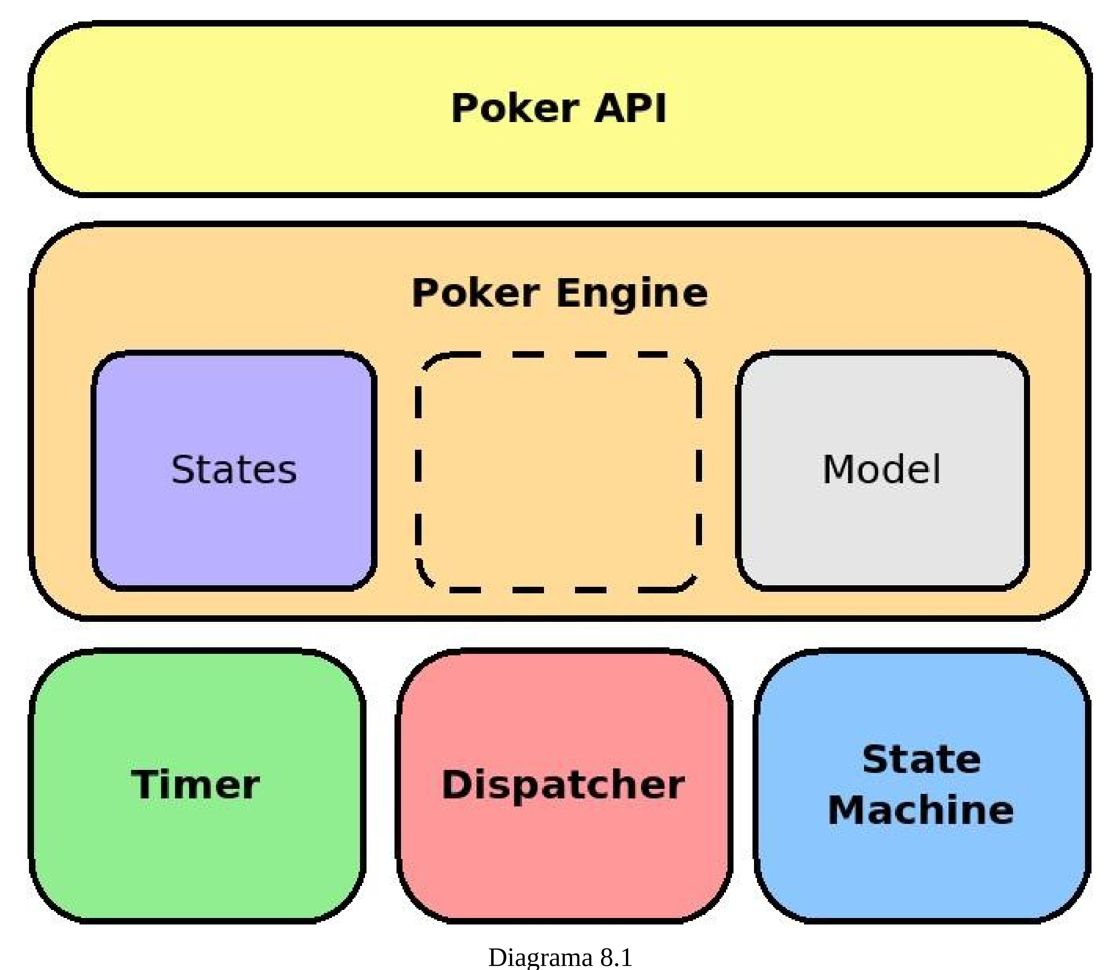


Diagrama 8.1

Como idea general, tenemos una representación de los distintos estados y como se interrelacionan en el Diagrama 8.2:

InitHand: El estado inicial de la máquina de estados y se ejecuta en el inicio de cada ronda.

BetRound: Estado de ronda de apuestas, cuando un jugador realiza su apuesta pasa a otro estado, aunque el nuevo estado sea el mismo.

Winner: El estado al que se pasa cuando sólo queda un jugador activo y ningún otro está en modo all-in, se trata de un caso muy particular.

Check: El estado en el que de forma general se pasa a la siguiente etapa en el juego, en las primeras etapas se van agregando las cartas comunitarias oportunas, hasta llegar a la etapa *ShowDown* y desde este estado se pasa al estado *ShowDown*.

ShowDown: Es el estado en el que determinan que jugadores ganan y cuánto.

EndHand: Es el último estado de la ronda, en él se determina cuántos jugadores quedan activos, si hay más de uno vuelve al *InitHand* y si no, finaliza la instancia de la máquina de estados.

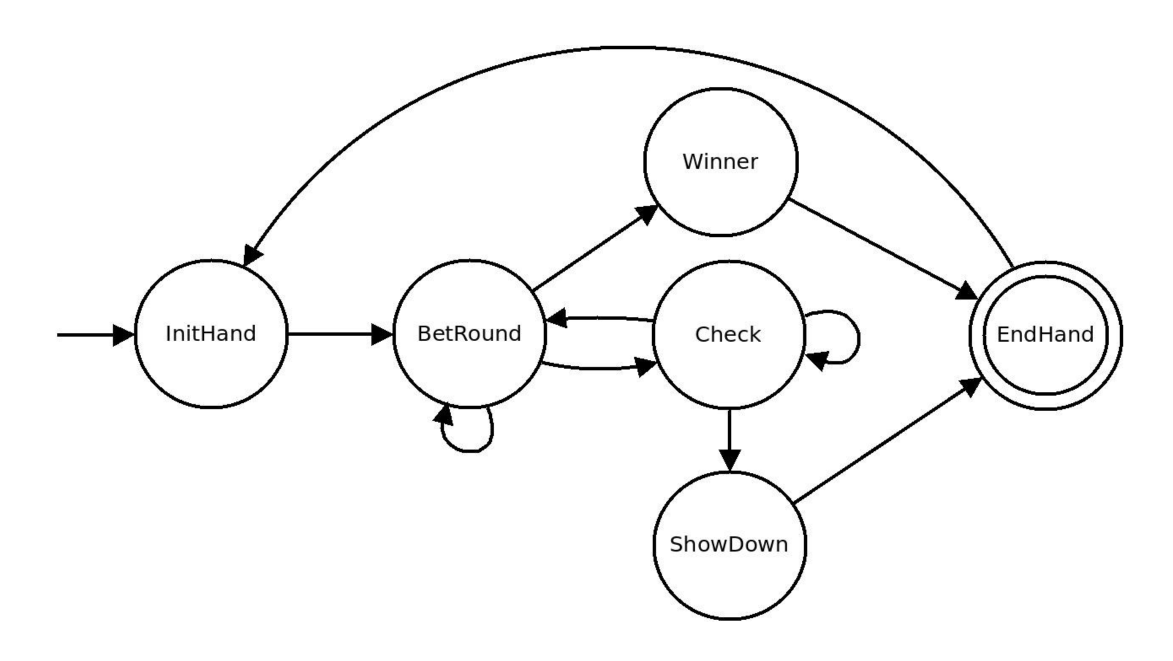


Diagrama 8.2

Como veis es relativamente simple, para convertir este diseño en código, vamos a modelar cada estado con una clase, de forma que quede recogida la lógica que habrá de ejecutar cuando una instancia de esta máquina de estados llegue a ese estado. Estos estados van a compartir la paquetería *org.poker.engine.states* y serán seis clases que implementen la interfaz definida en el capítulo 4 *IState<T>*, utilizando como parámetro '*T*' la clase definida en el capítulo anterior *ModelContext*: *IState<ModelContext>*.

Comenzamos por el estado inicial: *InitHandState* en el ejemplo 8.1.

```
public class InitHandState implements IState<ModelContext> {
    public static final String NAME = "InitHand";

    @Override
    public String getName() {
        return NAME;
    }
    @Override
    public boolean execute(ModelContext model) {
        Deck deck = model.getDeck();
        deck.shuffle();
        Settings settings = model.getSettings();
        model.setGameState(GameState.PRE_FLOP);
        model.clearCommunityCard();
        model.setRound(model.getRound() + 1);
        if (model.getRound() % settings.getRounds4IncrementBlind() == 0) {
            settings.setSmallBlind(2 * settings.getSmallBlind());
        }
        model.setPlayersAllIn(0);
        model.setHighBet(0L);
        List<PlayerEntity> players = model.getPlayers();
        for (PlayerEntity p : players) {
            p.setState(PlayerState.READY);
            p.setHandValue(0);
            p.setBet(0);
            p.showCards(false);
            p.setCards(deck.obtainCard(), deck.obtainCard());
        }
        int numPlayers = model.getNumPlayers();
        model.setActivePlayers(numPlayers);

        int dealerIndex = (model.getDealer() + 1) % numPlayers;
        model.setDealer(dealerIndex);
        model.setPlayerTurn((dealerIndex + 1) % numPlayers);
        if (numPlayers > MIN_PLAYERS) {
            compulsoryBet(model, settings.getSmallBlind());
        }
        compulsoryBet(model, settings.getBigBlind());
        return true;
    }
    private void compulsoryBet(ModelContext model, long chips) {
        int turn = model.getPlayerTurn();
        PlayerEntity player = model.getPlayer(turn);
        if (player.getChips() <= chips) {
            player.setState(PlayerState.ALL_IN);
            ModelUtil.playerBet(model, player, ALL_IN, player.getChips());
        } else {
            ModelUtil.playerBet(player, chips);
        }
        model.setHighBet(chips);
        model.setPlayerTurn((turn + 1) % model.getNumPlayers());
    }
}
```

Ejemplo 8.1

En el estado inicial, lo primero es mezclar la baraja e inicializar los atributos del juego como las cartas comunitarias, establecer el estado inicial, la apuesta más alta o los jugadores en estado all-in, se incrementa el número de la ronda y cuando superan cierto número (valor obtenido en la configuración inicial) se duplica el valor de las ciegas y se inicializan los parámetros de los jugadores activos, entregando las dos cartas individuales que tendrán para la ronda actual. Se actualiza el *Dealer* al jugador de su izquierda y se realizan las apuestas ciegas, que siempre que haya más de dos jugadores serán la ciega pequeña y la ciega grande a partir del jugador *Dealer*, estas apuestas son ciegas y obligatorias, por este motivo hay un método llamado *compulsoryBet*(), para realizar esta tarea. Si únicamente quedasen dos jugadores, se saltaría la ciega pequeña y el jugador que no tenga el rol de *Dealer* tendrá que realizar la ciega grande de forma obligatoria.

En el ejemplo 8.2 vamos a ver la implementación del estado *CheckState*.

```
public class CheckState implements IState<ModelContext> {
    public static final String NAME = "Next";
    private static final GameState[] GAME_STATE = GameState.values();
    private static final int[] OBATIN_CARDS = {3, 1, 1, 0, 0};

    public String getName() {
        return NAME;
    }
    private int indexByGameState(GameState gameState) {
        int i = 0;
        while (i < GAME_STATE.length && GAME_STATE[i] != gameState) {
            i++;
        }
        return i;
    }

    public boolean execute(ModelContext model) {
        int indexGameState = indexByGameState(model.getGameState());
        if (OBATIN_CARDS[indexGameState] > 0){
            model.addCommunityCards(OBATIN_CARDS[indexGameState]);
        }
        model.setGameState(GAME_STATE[indexGameState+1]);
        model.setLastActivePlayers(model.getActivePlayers());
        model.setBets(0);
        model.getPlayers().stream().filter(p -> p.isActive())
                          .forEach(p-> p.setState(READY));
        model.setPlayerTurn(ModelUtil.nextPlayer(model, model.getDealer()));
        model.setLastBetCommand(null);
        model.setLastPlayerBet(null);
        return true;
    }
}
```

Ejemplo 8.2

Aquí se agregan cartas comunitarias cuando resulta oportuno y se prepara el modelo para una nueva ronda de apuestas o para el siguiente estado *ShowDown*. Se utilizan constantes para tener los estados a mano y el número de cartas a agregar al conjunto de cartas comunitarias.

En los ejemplos 8.3a, 8.3b y 8.3c veremos cómo queda la implementación del estado *BetRoundState*.

```
public class BetRoundState implements IState<ModelContext> {

    @FunctionalInterface
    private static interface BetChecker {

        public boolean check(ModelContext m, PlayerEntity player, BetCommand bet);
    }

    public static final String NAME = "BetRound";
    private static final Map<BetCommandType, BetChecker> CHECKERS =
                                                        buildBetCommandChecker();

    @Override
    public String getName() {
        return NAME;
    }

    private static Map<BetCommandType, BetChecker> buildBetCommandChecker() {
        Map<BetCommandType, BetChecker> result = new EnumMap<>(BetCommandType.class);
        result.put(BetCommandType.FOLD, (m, p, b) -> true);
        result.put(BetCommandType.TIMEOUT, (m, p, b) -> false);
        result.put(BetCommandType.ERROR, (m, p, b) -> false);

        result.put(BetCommandType.RAISE, (m, p, b) ->
             b.getChips() > (m.getHighBet() - p.getBet()) &&
             b.getChips() < p.getChips());

        result.put(BetCommandType.ALL_IN, (m, p, b) -> {
            b.setChips(p.getChips());
            return p.getChips() > 0;
        });

        result.put(BetCommandType.CALL, (c, p, b) -> {
            b.setChips(c.getHighBet() - p.getBet());
            return c.getHighBet() > c.getSettings().getBigBlind();
        });

        result.put(BetCommandType.CHECK, (c, p, b) -> {
            b.setChips(c.getHighBet() - p.getBet());
            return b.getChips() == 0|| c.getHighBet()==c.getSettings().getBigBlind();
        });
        return result;
    }
    ...
```

Ejemplo 8.3a

En esta primera parte vemos como hemos definido una interfaz estática llamada *BetChecker*, con un único método que nos va a permitir utilizar expresiones lambda, después tenemos una constante, `CHECKERS`, nos servirá para obtener un *BetChecker*, que nos proporcionará el mecanismo para comprobar si un comando enviado por el jugador es correcto o es erróneo.

```
...
private static Map<BetCommandType, BetChecker> buildBetCommandChecker() {
    Map<BetCommandType, BetChecker> result = new EnumMap<>(BetCommandType.class);
    result.put(BetCommandType.FOLD, (m, p, b) -> true);
    result.put(BetCommandType.TIMEOUT, (m, p, b) -> false);
    result.put(BetCommandType.ERROR, (m, p, b) -> false);

    result.put(BetCommandType.RAISE, (m, p, b) ->
         b.getChips() > (m.getHighBet() - p.getBet()) &&
         b.getChips() < p.getChips());

    result.put(BetCommandType.ALL_IN, (m, p, b) -> {
        b.setChips(p.getChips());
        return p.getChips() > 0;
    });

    result.put(BetCommandType.CALL, (c, p, b) -> {
        b.setChips(c.getHighBet() - p.getBet());
        return c.getHighBet() > c.getSettings().getBigBlind();
    });

    result.put(BetCommandType.CHECK, (c, p, b) -> {
        b.setChips(c.getHighBet() - p.getBet());
        return b.getChips()==0 || c.getHighBet()==c.getSettings().getBigBlind();
    });
    return result;
}
...
```

Ejemplo 8.3b

En el método *buildBetCommandChecker* se construye otra constante, esta aglutinará los BetChecker que vamos a utilizar para validar si la terna *ModelContext*, *PlayerEntity* y *BetCommand* es una apuesta valida. Estos *BetChecker* están implementados como expresiones Lambda e indexadas por el tipo de *BetComandType*, por este motivo tiene la anotación *@FunctionalInterface*.

Todas las apuestas de tipo *Fold* son válidas, las apuestas de time-out o de error son erróneas y con respecto al resto dependerá de la situación de la partida, Para apostar todo hace falta apostar todas las fichas disponibles, para elevar la apuesta hace falta tener fichas disponibles y que la apuesta sea más alta que la anterior y que no sean todas las fichas del jugador, para un comando de tipo *Call* es necesario igualar la apuesta, y para *check* que no haga falta apostar para continuar o que únicamente haya que igualar la ciega grande.

Como vemos, ya estamos empezando a sacar partido a las expresiones Lambda. Evitamos crear varias clases anónimas manteniendo la claridad en el código.

```
    ...
    public boolean execute(ModelContext model) {
        boolean result = false;
        int playerTurn = model.getPlayerTurn();
        PlayerEntity player = model.getPlayer(playerTurn);
        BetCommand command = player.getBetCommand();
        if (command != null) {
            BetCommand resultCommand = command;
            player.setBetCommand(null);
            long betChips = 0;
            BetCommandType commandType = command.getType();
            if (CHECKERS.get(commandType).check(model, player, command)) {
                betChips = command.getChips();
                player.setState(TexasHoldEmUtil.convert(command.getType()));
            } else {
                commandType = BetCommandType.FOLD;
                player.setState(PlayerState.FOLD);
                if (command.getType() == BetCommandType.TIMEOUT) {
                    resultCommand = new BetCommand(BetCommandType.TIMEOUT);
                } else {
                    resultCommand = new BetCommand(BetCommandType.ERROR);
                }
                ModelUtil.incrementErrors(player, model.getSettings());
            }
            ModelUtil.playerBet(model, player, commandType, betChips);
            model.lastResultCommand(player, resultCommand);
            model.setPlayerTurn(ModelUtil.nextPlayer(model, playerTurn));
            result = true;
        }
        return result;
    }
}
```

Ejemplo 8.3c

El método *execute* forma parte de la interfaz de *IState*, aquí se utilizarán todos los elementos antes descritos para verificar la apuesta e integrarla en el modelo compartido. Si no hubiera una apuesta, se devolverá *false* para indicar que el estado no puede avanzar y debe esperar a algún evento. En caso de una apuesta errónea, el jugador pasa y si excede el número máximo de errores abandonará la partida, si la apuesta es válida, se incorpora al modelo y se actualiza la información oportuna, en ambos casos, el método devuelve *true* para indicar que hay que continuar en el flujo de la máquina de estados.

Este estado es el más complejo que tenemos en el juego, ya que espera recibir un evento, bien de un jugador o bien del temporizador y la lógica debe contemplar la casuística de una ejecución inicial: la que se produce para indicar que se espera un evento, afortunadamente es el único punto de interacción que hay entre los jugadores y la máquina de estados del juego.

En el ejemplo 8.4 veremos cómo queda la implementación del estado *WinnerState*.

```
public class WinnerState implements IState<ModelContext> {

    public static final String NAME = "Winner";

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public boolean execute(ModelContext model) {
        List<PlayerEntity> players = model.getPlayers();
        players.stream()
                .filter(p -> p.isActive() || p.getState() == PlayerState.ALL_IN)
                .findFirst()
                .get()
                .addChips(players
                        .stream()
                        .mapToLong(p -> p.getBet())
                        .sum());
        return true;
    }
}
```

Ejemplo 8.4

En el método *execute()* de *WinnerState* vamos a emplear otra de las novedades de Java 8: los *Stream*.

En este caso en concreto, se filtran los jugadores activos o que han apostado todo y luego se queda con el primero (y único que debería cumplir con la condición del filtro), a ese jugador se le agregan todas las fichas apostadas, para calcular este valor se vuelve a utilizar la funcionalidad del *Stream*, pero esta vez se realiza un mapeo, se convierte un *PlayerEntity* en un *long*, el modo utilizado es devolviendo por cada instancia de *PlayerEntity* las fichas apostadas, en un segundo paso se suman estos valores, eso se hace con la función sum(), pero esta función se ejecuta sobre un *Stream* de *int* o de *long*, por este motivo se hace un *mapToLong* en lugar de *map*.

El método devuelve *true* para continuar con el flujo de la máquina de estados.

Los Stream vienen con funcionalidad muy interesante, admiten filtros, colectores, agrupadores, mapeo de un tipo a otro, saltos de elementos, ejecutar acciones en medio, ordenar elementos, eliminar duplicados, reducir el flujo, etc... En el siguiente estado *ShowDownState* vamos a tratar de sacar más partido.

En el ejemplo 8.5a y 8.5b veremos cómo queda la implementación del estado ShowDownState.

```
public class ShowDownState implements IState<ModelContext> {

    public static final String NAME = "ShowDown";

    @Override
    public String getName() {
        return NAME;
    }

    private List<PlayerEntity> calculateHandValue(List<Card> cc,
                                              List<PlayerEntity> players) {
        Hand7Evaluator evaluator = new Hand7Evaluator(new HandEvaluator());
        evaluator.setCommunityCards(cc);
        players.stream().forEach(
            p -> p.setHandValue(evaluator.eval(p.getCard(0), p.getCard(1))));
        return players;
    }

    public boolean execute(ModelContext model) {
        List<PlayerEntity> players =
                calculateHandValue(model.getCommunityCards(), model.getPlayers());
        Map<Long, List<PlayerEntity>> indexByBet = players.stream()
                .filter(p -> p.getBet() > 0)
                .collect(Collectors.groupingBy(p -> p.getBet()));
        List<Long> inverseSortBets = new ArrayList<>(indexByBet.keySet());
        Collections.sort(inverseSortBets, (l0, l1) -> Long.compare(l1, l0));

        Iterator<Long> it = inverseSortBets.iterator();
        List<PlayerEntity> lastPlayers = indexByBet.get(it.next());
        while (it.hasNext()) {
            List<PlayerEntity> currentPlayers = indexByBet.get(it.next());
            currentPlayers.addAll(lastPlayers);
            lastPlayers = currentPlayers;
        }
    ...
```

Ejemplo 8.5a

Este estado es el más complejo y en el que vamos a desplegar la gran potencia de los *Stream*, en primer lugar tenemos un método privado que calcula el valor de las manos de cada jugador apoyándose en la clase Hand7Evaluator del capítulo 3.

A continuación tenemos la implementación del método *execute()*, de modo que en primer lugar se obtienen la lista de jugadores después de haber evaluado sus manos. Tras esto se agrupan los jugadores según la cuantía de su apuesta, de tal modo en la variable *indexByBet* por cada cantidad de fichas apostadas tenemos una lista con los jugadores que realizaron esa apuesta. Tras esto vamos a recorrer la lista de jugadores en orden inverso según su apuesta, empezando por los jugadores con la apuesta más alta para ir agregándolos a las listas de jugadores con una apuesta inferior.

```
        ...
        Set<Long> bet4Analysis = players.stream()
                .filter(p -> p.getState() == PlayerState.ALL_IN)
                .map(p -> p.getBet()).collect(Collectors.toSet());
        bet4Analysis.add(inverseSortBets.get(0));
        long accumulateChips = 0L;
        long lastBet = 0L;
        while (!inverseSortBets.isEmpty()) {
            Long bet = inverseSortBets.remove(inverseSortBets.size() - 1);
            List<PlayerEntity> currentPlayers = indexByBet.get(bet);
            accumulateChips += (bet - lastBet) * currentPlayers.size();
            if (bet4Analysis.contains(bet)) {
                Collections.sort(currentPlayers,
                                 (p0, p1) -> p1.getHandValue() - p0.getHandValue());
                List<PlayerEntity> winners = new ArrayList<>(currentPlayers.size());
                currentPlayers.stream()
                        .filter(p -> p.getState() != PlayerState.OUT)
                        .peek  (p -> p.showCards(true))
                        .filter(p -> winners.isEmpty() ||
                                p.getHandValue() == winners.get(0).getHandValue())
                        .forEach(winners::add);
                long chips4Player = accumulateChips / winners.size();
                winners.stream().forEach(p -> p.addChips(chips4Player));
                int remain = (int) accumulateChips % winners.size();
                if (remain > 0) {
                    Collections.shuffle(winners);
                    winners.stream().limit(remain).forEach(p -> p.addChips(1));
                }
                accumulateChips = 0L;
            }
            lastBet = bet;
        }
        return true;
    }
}
```

Ejemplo 8.5b

Tras este paso, vamos a determinar los botes disponibles, se calcularán a partir de las cantidades de los jugadores han apostado todo (All-in) y del que ha hecho la apuesta más alta.

Se calculará para cada bote disponible, cuál de los jugadores que compiten para ganarlo tiene la mejor mano y que no haya abandonado la ronda, ordenándolos de mayor a menor por el valor de sus manos, en caso de igualdad, se reparten las fichas a partes iguales y si en el reparto no hay una división exacta, las fichas que quedan se sortean entre los ganadores de ese bote.

El método devuelve *true* para que fluya el proceso en la máquina de estados. Aquí termina la implementación de la lógica asociada a este estado, densa en complejidad, pero sin embargo no es muy extensa en líneas gracias a las nuevas funcionalidades llegadas con *Stream*.

Por supuesto hay otros modos de llevar a cabo este algoritmo y seguramente sean mucho más sencillos, un poco más adelante tendremos la oportunidad de hacer algo parecido de otro modo.

En el ejemplo 8.6 veremos cómo queda la implementación del estado *EndState*.

```
public class EndState implements IState<ModelContext> {

    public static final String NAME = "EndHand";

    @Override
    public String getName() {
        return NAME;
    }

    @Override
    public boolean execute(ModelContext model) {
        PlayerEntity dealerPlayer = model.getPlayer(model.getDealer());
        List<PlayerEntity> players = model.getPlayers();
        List<PlayerEntity> nextPlayers = new ArrayList<>(players.size());
        int i = 0;
        int dealerIndex = 0;
        for (PlayerEntity p : players) {
            if (p.getChips() > 0) {
                nextPlayers.add(p);
                i++;
            }
            if (dealerPlayer == p) {
                dealerIndex = i-1;
            }
        }
        model.setDealer(dealerIndex);
        model.setPlayers(nextPlayers);
        return true;
    }
}
```

Ejemplo 8.6

La lógica asociada al final de la mano simplemente actualiza el *Dealer* y elimina los jugadores que se han quedado fuera del juego en la ronda que concluye.

Aquí damos por finalizado el capítulo 8, en el diagrama 8.2 hemos visto cómo se relacionan los distintos estados, sin embargo en las implementaciones no hay nada que haga referencia a esas relaciones, en el siguiente capítulo veremos cómo se integran.

Capítulo 9: Controladores para el Póquer

En este capítulo vamos a ver el módulo de los controladores y adaptadores de la máquina de estados del Póquer, estas clases son las que permitirán integrar la lógica del juego y velarán por que las estrategias cumplan con las reglas del juego.

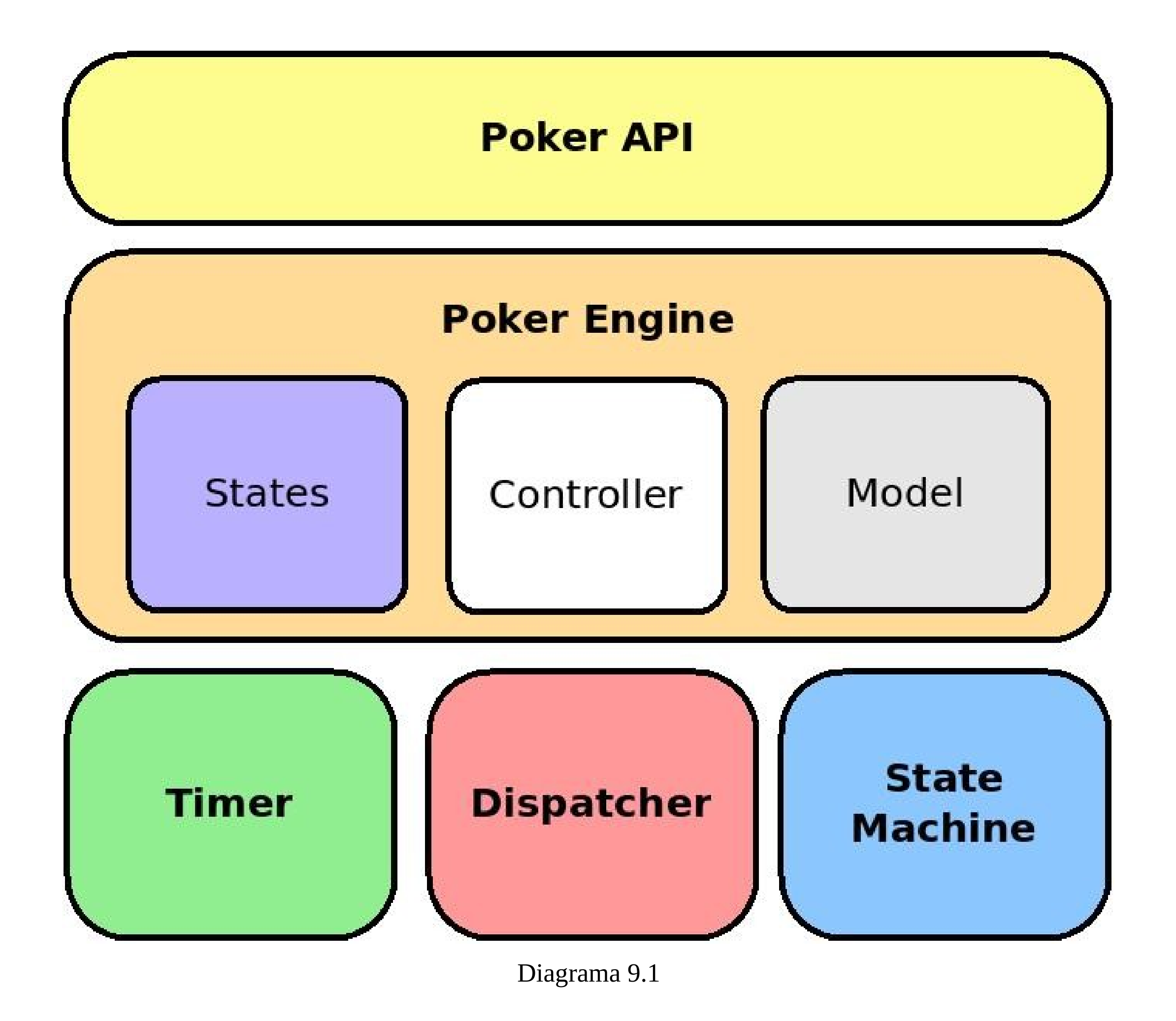


Diagrama 9.1

Comenzamos con una clase de utilidades para adaptar la información de la lógica interna a la lógica que manejará la API del juego. Esta clase de utilidades la denominaremos *PlayerAdapter* que podemos ver en el ejemplo 9.1

```
public final class PlayerAdapter {

    private PlayerAdapter() {
    }

    public static List<PlayerInfo> toPlayerInfo(Collection<PlayerEntity> players,
                                                String name) {
        List<PlayerInfo> result = Collections.emptyList();
        if (players != null) {
            result = new ArrayList<>(players.size());
            for (PlayerEntity pe : players) {
                result.add(copy(pe, pe.showCards() || pe.getName().equals(name)));
            }
        }
        return result;
    }

    public static GameInfo<PlayerInfo> toTableState(ModelContext model, String name) {
        GameInfo<PlayerInfo> result = new GameInfo<>();
        result.setCommunityCards(model.getCommunityCards());
        result.setDealer(model.getDealer());
        result.setGameState(model.getGameState());
        result.setPlayerTurn(model.getPlayerTurn());
        result.setRound(model.getRound());
        if (model.getSettings() != null) {
            result.setSettings(new Settings(model.getSettings()));
        }
        result.setPlayers(toPlayerInfo(model.getPlayers(), name));
        return result;
    }

    public static PlayerInfo copy(PlayerEntity p, boolean copyCards) {
        PlayerInfo result = new PlayerInfo();
        result.setName(p.getName());
        result.setChips(p.getChips());
        result.setBet(p.getBet());
        if (copyCards) {
            result.setCards(p.getCard(0), p.getCard(1));
        }
        result.setState(p.getState());
        result.setErrors(p.getErrors());
        return result;
    }
}
```

Ejemplo 9.1

Ahora toca crear la clase *StateMachineConnector*, que como su nombre indica se trata de una clase para conectar con una instancia de la máquina de estados del juego del Póquer, los eventos que se produzcan ya sea por el comportamiento de los jugadores o por el del temporizador se canalizaran a través de esta clase para hacerlos llegar a la instancia de la máquina de estados. Además esta clase será la responsable de instanciar y definir la máquina de estados. En los ejemplos 9.2a, 9.2b y 9.2c tenemos la implementación de la clase *StateMachineConnector.*

```
public class StateMachineConnector {
    private static final Logger LOGGER = LoggerFactory.getLogger(State..class);

    private static final int END_HAND_SLEEP_TIME = 1000;
    public static final String NEXT_PLAYER_TURN = "nextPlayerTurn";
    private final StateMachine<ModelContext> texasStateMachine = buildStateMachine();
    private final Map<String, IGameEventDispatcher> playersDispatcher;
    private final IGameTimer timer;
    private ModelContext model;
    private IGameEventDispatcher system;
    private StateMachineInstance<ModelContext> instance;
    private long timeoutId = 0;

    public StateMachineConnector(IGameTimer t,Map<String,IGameEventDispatcher> pd){
        this.playersDispatcher = pd;
        this.timer = t;
    }

    public void setSystem(IGameEventDispatcher system) {
        this.system = system;
    }
    public void createGame(Settings settings) {
        if (model == null) {
            LOGGER.debug("createGame: {}", settings);
            model = new ModelContext(settings);
            model.setDealer(-1);
        }
    }

    public void addPlayer(String playerName) {
        if (model != null) {
            LOGGER.debug("addPlayer: \"{}\"", playerName);
            model.addPlayer(playerName);
        }
    }
    public void startGame() {
        LOGGER.debug("startGame");
        if (instance == null && model != null) {
            model.setDeck(new Deck());
            instance = texasStateMachine.startInstance(model);
            model.setDealer(0);
            execute();
        }
    }
    ...
```

Ejemplo 9.2a

```
...
public void betCommand(String playerName, BetCommand command) {
    LOGGER.debug("betCommand: {} -> {}", playerName, command);
    if (instance != null && playerName.equals(model.getPlayerTurnName())) {
        BetCommand betCommand = command;
        if (betCommand == null) {
            betCommand = new BetCommand(TexasHoldEmUtil.BetCommandType.ERROR);
        }
        model.getPlayerByName(playerName).setBetCommand(betCommand);
        execute();
    }
}

public void timeOutCommand(Long timeoutId) {
    LOGGER.debug("timeOutCommand: id: {}", timeoutId);
    if (instance != null && timeoutId == this.timeoutId) {
        LOGGER.debug("timeOutCommand: player: {}", model.getPlayerTurnName());
        model.getPlayerByName(model.getPlayerTurnName()).setBetCommand(
                         new BetCommand(TexasHoldEmUtil.BetCommandType.TIMEOUT));
        execute();
    }
}

private void execute() {
    if (instance.execute().isFinish()) {
        model.setGameState(TexasHoldEmUtil.GameState.END);
        model.setCommunityCards(Collections.emptyList());
        notifyEndGame();
        instance = null;
    }
}

private void notifyInitHand() {
    notifyEvent(GameController.INIT_HAND_EVENT_TYPE);
}
private void notifyBetCommand() {
    String playerTurn = model.getLastPlayerBet().getName();
    BetCommand lbc = model.getLastBetCommand();
    LOGGER.debug("notifyBetCommand -> {}: {}", playerTurn, lbc);
    for (String playerName : playersDispatcher.keySet()) {
        playersDispatcher.get(playerName).dispatch(
            new GameEvent(GameController.BET_COMMAND_EVENT_TYPE, playerTurn,
                           new BetCommand(lbc.getType(), lbc.getChips())));
    }
}

private void notifyCheck() {
    LOGGER.debug("notifyCheck: {}", GameController.CHECK_PLAYER_EVENT_TYPE...);
    for (String playerName : playersDispatcher.keySet()) {
        playersDispatcher.get(playerName).dispatch(
            new GameEvent(GameController.CHECK_PLAYER_EVENT_TYPE,
                SYSTEM_CONTROLLER, model.getCommunityCards()));
    }
}
...
```

Ejemplo 9.2b

```
...
private void notifyPlayerTurn() {
    String playerTurn = model.getPlayerTurnName();
    if (playerTurn != null) {
        LOGGER.debug("notifyPlayerTurn -> {}", playerTurn);
        playersDispatcher.get(playerTurn).dispatch(
            new GameEvent(GameController.GET_COMMAND_PLAYER_EVENT_TYPE,
                SYSTEM_CONTROLLER,
                PlayerAdapter.toTableState(model, playerTurn)));
    }
    timer.resetTimer(++timeoutId);
}

private void notifyEndHand() {
    notifyEvent(GameController.END_HAND_PLAYER_EVENT_TYPE);
    try {
        Thread.sleep(END_HAND_SLEEP_TIME);
    } catch (InterruptedException ex) {
        LOGGER.error("Error en la espera despues de terminar una mano.", ex);
    }
}

private void notifyEndGame() {
    notifyEvent(GameController.END_GAME_PLAYER_EVENT_TYPE);
    system.dispatch(
        new GameEvent(GameController.EXIT_CONNECTOR_EVENT_TYPE,
                      SYSTEM_CONTROLLER));
    notifyEvent(GameController.EXIT_CONNECTOR_EVENT_TYPE);
}

private void notifyEvent(String type) {
    LOGGER.debug("notifyEvent: {} -> {}", type, model);
    for (String playerName : playersDispatcher.keySet()) {
        playersDispatcher.get(playerName).dispatch(
                new GameEvent(type, SYSTEM_CONTROLLER,
                    PlayerAdapter.toTableState(model, playerName)));
    }
}
...
```

Ejemplo 9.2c

Los métodos que van desde el *createGame()* del principio (ejemplo 9.2a) hasta el método *execute* (ejemplo 9.2b) son métodos de entrada de eventos. El método *execute()* trata de validar si la instancia de la máquina de estados llega al final para notificar el fin de la partida. Desde el siguiente método al *execute()* hasta el *notifyEvent* (ejemplo 9.2c) son métodos para notificar los eventos producidos en el juego.

El método *buildStateMachine* (ejemplo 9.2d, en la siguiente página), construye una máquina de estados con toda la funcionalidad recogida en los capítulos anteriores, define las transiciones entre estados y la lógica a ejecutar en cada uno de ellos así como lógica extra para ejecutar al llegar a un estado o al abandonarlo. El resultado podría ser más compacto, sin embargo y en pos de la legibilidad he dejado líneas en blanco para separar ciertos bloques de código.

```
    ...
    private StateMachine<ModelContext> buildStateMachine() {
        StateMachine<ModelContext> sm = new StateMachine<>();
        final IState<ModelContext> initHandState =
            StateDecoratorBuilder.after(new InitHandState(), () -> notifyInitHand());

        final IState<ModelContext> betRoundState = StateDecoratorBuilder
                .create(new BetRoundState())
                .before(() -> notifyPlayerTurn())
                .after(() -> notifyBetCommand())
                .build();

        final IState<ModelContext> checkState =
            StateDecoratorBuilder.after(new CheckState(), () -> notifyCheck());
        final IState<ModelContext> showDownState = new ShowDownState();
        final IState<ModelContext> winnerState = new WinnerState();
        final IState<ModelContext> endHandState =
            StateDecoratorBuilder.before(new EndState(), () -> notifyEndHand());

        sm.setInitState(initHandState);

        // initHandState transitions
        sm.setDefaultTransition(initHandState, betRoundState);

        // betRoundState transitions
        sm.addTransition(betRoundState, betRoundState,
            c -> c.getPlayerTurn() != ModelUtil.NO_PLAYER_TURN);
        sm.addTransition(betRoundState, winnerState,
            c -> c.getPlayersAllIn() + c.getActivePlayers() == 1);
        sm.setDefaultTransition(betRoundState, checkState);

        // checkState transitions
        sm.addTransition(checkState, showDownState,
            c -> c.getGameState() == TexasHoldEmUtil.GameState.SHOWDOWN);
        sm.addTransition(checkState, betRoundState,
                                   c -> c.getPlayerTurn() != ModelUtil.NO_PLAYER_TURN);
        sm.setDefaultTransition(checkState, checkState);

        // betWinnerState transitions
        sm.setDefaultTransition(winnerState, endHandState);

        // showDownState transitions
        sm.setDefaultTransition(showDownState, endHandState);

        // endHandState transitions
        sm.addTransition(endHandState, initHandState, c -> c.getNumPlayers() > 1
                                       && c.getRound() < c.getSettings().getMaxRounds());
        return sm;
    }
}
```

Ejemplo 9.2d

La clase *GameController* es la que realiza la mayor parte de la 'fontanería' interna, conecta las estrategias de los jugadores con el conector de la máquina de estados y al temporizador, prepara los grupos de hilos, comprueba la configuración e inicia el juego, una vez que este termina, este controlador libera todos los hilos implicados en él.

La vemos en los ejemplos 9.3a, 9.3b y 9.3c.

```
public class GameController implements IGameController, Runnable {

    private static final Logger LOGGER = LoggerFactory.getLogger(GameContr...class);
    private static final int DISPATCHER_THREADS = 1;
    private static final int EXTRA_THREADS = 2;
    public static final String SYSTEM_CONTROLLER = "system";
    public static final String INIT_HAND_EVENT_TYPE = "initHand";
    public static final String BET_COMMAND_EVENT_TYPE = "betCommand";
    public static final String END_GAME_PLAYER_EVENT_TYPE = "endGame";
    public static final String END_HAND_PLAYER_EVENT_TYPE = "endHand";
    public static final String CHECK_PLAYER_EVENT_TYPE = "check";
    public static final String GET_COMMAND_PLAYER_EVENT_TYPE = "getCommand";
    public static final String EXIT_CONNECTOR_EVENT_TYPE = "exit";
    public static final String ADD_PLAYER_CONNECTOR_EVENT_TYPE = "addPlayer";
    public static final String TIMEOUT_CONNECTOR_EVENT_TYPE = "timeOutCommand";
    public static final String CREATE_GAME_CONNECTOR_EVENT_TYPE = "createGame";
    private final Map<String, IGameEventDispatcher> players = new HashMap<>();
    private final List<String> playersByName = new ArrayList<>();
    private final Map<String, IGameEventProcessor<IStrategy>> playerProcessors;
    private final GameEventDispatcher<StateMachineConnector> connectorDispatcher;
    private final StateMachineConnector stateMachineConnector;
    private final IGameTimer timer;
    private Settings settings;
    private ExecutorService executors;
    private List<ExecutorService> subExecutors = new ArrayList<>();

    public GameController() {
        timer = new GameTimer(SYSTEM_CONTROLLER, buildExecutor(DISPATCHER_THREADS));
        stateMachineConnector = new StateMachineConnector(timer, players);
        connectorDispatcher = new GameEventDispatcher<>(stateMachineConnector,
                                                        buildConnectorProcessors(),
                                                        buildExecutor(1));
        stateMachineConnector.setSystem(connectorDispatcher);
        timer.setDispatcher(connectorDispatcher);
        playerProcessors = buildPlayerProcessors();
    }

    private ExecutorService buildExecutor(int threads){
        ExecutorService result = Executors.newFixedThreadPool(threads);
        subExecutors.add(result);
        return result;
    }
    @Override
    public void setSettings(Settings settings) {
        this.settings = new Settings(settings);
    }
    ...
```

Ejemplo 9.3a

```
...
private static Map<String, IgameEventProcessor<StateMachineConnector>>
                                                    buildConnectorProcessors() {
    Map<String, IGameEventProcessor<StateMachineConnector>> cpm =new HashMap<>();
    cpm.put(CREATE_GAME_CONNECTOR_EVENT_TYPE,
        (c, e) -> c.createGame((Settings) e.getPayload()));
    cpm.put(ADD_PLAYER_CONNECTOR_EVENT_TYPE,
        (c, e) -> c.addPlayer(e.getSource()));
    cpm.put(INIT_HAND_EVENT_TYPE, (c, e) -> c.startGame());
    cpm.put(BET_COMMAND_EVENT_TYPE,
        (c, e) -> c.betCommand(e.getSource(), (BetCommand) e.getPayload()));
    cpm.put(TIMEOUT_CONNECTOR_EVENT_TYPE,
        (c, e) -> c.timeOutCommand((Long) e.getPayload()));
    return cpm;
}

private Map<String, IGameEventProcessor<IStrategy>> buildPlayerProcessors() {
    Map<String, IGameEventProcessor<IStrategy>> ppm = new HashMap<>();
    IGameEventProcessor<IStrategy> defaultProcessor =
        (s, e) -> s.updateState((GameInfo) e.getPayload());
    ppm.put(INIT_HAND_EVENT_TYPE, defaultProcessor);
    ppm.put(END_GAME_PLAYER_EVENT_TYPE, defaultProcessor);
    ppm.put(BET_COMMAND_EVENT_TYPE,
        (s, e) -> s.onPlayerCommand(e.getSource(), (BetCommand) e.getPayload()));
    ppm.put(CHECK_PLAYER_EVENT_TYPE,
        (s, e) -> s.check((List<Card>) e.getPayload()));
    ppm.put(GET_COMMAND_PLAYER_EVENT_TYPE, (s, e) -> {
        GameInfo<PlayerInfo> gi = (GameInfo<PlayerInfo>) e.getPayload();
        String playerTurn = gi.getPlayers().get(gi.getPlayerTurn()).getName();
        BetCommand cmd = s.getCommand(gi);
        connectorDispatcher.dispatch(
            new GameEvent(BET_COMMAND_EVENT_TYPE, playerTurn, cmd));
    });
    return ppm;
}

public synchronized boolean addStrategy(IStrategy strategy) {
    boolean result = false;
    String name = strategy.getName();
    if (!players.containsKey(name) && !SYSTEM_CONTROLLER.equals(name)) {
        players.put(name, new GameEventDispatcher<>(strategy,
            playerProcessors, buildExecutor(DISPATCHER_THREADS)));
        playersByName.add(name);
        result = true;
    }
    return result;
}

public synchronized void waitFinish() {
    if (!finish) {
        try {wait();}
        catch (InterruptedException ex) {LOGGER.error("Esperando el...", ex);}
    }
}
...
```

Ejemplo 9.3b

```
    ...
    private void check(boolean essentialCondition, String exceptionMessage)
                                                            throws GameException {
        if (!essentialCondition) {
            throw new GameException(exceptionMessage);
        }
    }
    @Override
    public synchronized void start() throws GameException {
        LOGGER.debug("start");
        check(settings != null, "No se ha establecido una configuración.");
        check(players.size() > 1, "No se han agregado un numero suficiente de j...);
        check(players.size() <= settings.getMaxPlayers(), "El número de jugador...);
        check(settings.getMaxErrors() > 0, "El número de máximo de errores debe...);
        check(settings.getMaxRounds() > 0, "El número de máximo de rondas debe ...);
        check(settings.getRounds4IncrementBlind()> 1, "El número de rondas hast...);
        check(settings.getTime() > 0, "El tiempo másimo por jugador debe ser ma...);
        check(settings.getPlayerChip() > 0, "El número de fichas inicial por ju...);
        check(settings.getSmallBlind() > 0, "La apuesta de la ciega pequeña debe..);
        executors = Executors.newFixedThreadPool(players.size() + EXTRA_THREADS);
        players.values().stream().forEach(executors::execute);
        stateMachineConnector.createGame(settings);
        timer.setTime(settings.getTime());
        playersByName.stream().forEach(stateMachineConnector::addPlayer);
        executors.execute(timer);
        finish = false;
        new Thread(this).start();
    }
    @Override
    public synchronized void run() {
        LOGGER.debug("run");
        // Inicio de la ejecución del juego.
        connectorDispatcher.dispatch(
            new GameEvent(INIT_HAND_EVENT_TYPE, SYSTEM_CONTROLLER));
        connectorDispatcher.run();
        // Fin de la ejecución del juego.
        timer.exit();
        executors.shutdown();
        players.values().stream().forEach(IGameEventDispatcher::exit);
        subExecutors.stream().forEach(ExecutorService::shutdown);
        try {
            executors.awaitTermination(0, TimeUnit.SECONDS);
        } catch (InterruptedException ex) {
            LOGGER.error("Error intentando eliminar cerrar todos los hilos", ex);
        }
        finish = true;
        notifyAll();
    }
}
```

Ejemplo 9.3c

Como ya hemos comentado, esta clase hace de puente entre las estrategias de los jugadores, el conector de la máquina de estados y el temporizador. Las estrategias y el conector de la máquina de estados los gestiona mediante un *GameEventDispatcher.*

Esto se debe a una razón muy simple, se pretende que estos elementos se ejecuten de forma aislada, que toda la ejecución del *GameEventDispatcher* quede dentro de hilos independientes y que los procesados de este se realicen también de forma independiente. Como el grupo de hilos para el conector tiene únicamente un elemento, toda la ejecución de su lógica se realizará siempre en el mismo hilo y por tanto no hemos tenido que preocuparnos de la sincronización de los métodos del mismo. Sin embargo, el grupo de hilos de las estrategias podemos querer gestionarlo de forma diferente, aunque en este caso tiene un único hilo, podríamos configurar varios o podríamos configurar el mismo grupo de hilos para todos los componentes, están en grupos separados para que ningún componente pueda acaparar todos los hilos del grupo general.

En el método *start()* se comprueban una serie de condiciones y se eleva una excepción propia *GameException*, dicha excepción no la hemos comentado en ningún capítulo anterior, su lugar natural debería ser la API del juego, en la paquetería *org.poker.api.game* y es aquí donde la vamos a dejar. En el ejemplo 9.4 tenemos su implementación.

```
public class GameException extends Exception {
    public GameException(String message) {
        super(message);
    }
}
```

Ejemplo 9.4

La excepción *GameException* hereda de la clase *Exception* y únicamente sobrescribe un constructor, esto significa que nuestra excepción será comprobada, por lo que el método *start()* de la clase *GameController* tiene que indicar que puede lanzar dicha excepción, por tanto también hay que modificar la interfaz *IGameController* para incluir en la definición del método la posibilidad de lanzar GameException, de no hacerlo el compilador nos devolverá un error.

La consecuencia de lanzar una excepción comprobada es que el código que invoque al método *start()* de la clase *GameController* deberá gestionar esa posibilidad, una opción es realizar la invocación dentro de una sentencia try-catch o try-catch-finally, otra es indicar que el método desde donde se ejecuta puede lanzar esa excepción y la última posibilidad es lanzar otro tipo de excepción que indique a esta como causa, con la posibilidad de que la nueva excepción pueda ser no comprobada.

Si la excepción fuera no comprobada, iría subiendo hasta encontrarse con una sentencia try-catch/ try-catch-finally que pueda capturarla, en caso de no encontrar ninguna que recoja esta excepción o alguna superclase de la misma, llegaría hasta arriba acabando con el hilo en el que se produjo y en caso de ser el último hilo, el proceso finalizaría con ese error. Esta situación también podría llegar a darse cuando la excepción es comprobada, pero no es muy común que ocurra.

En el capítulo 7 de **“Clean Code a Handbook of Agile Software”** hay una buena base para comenzar con la gestión de errores y el uso de excepciones.

Capítulo 10: Creando la estrategia del jugador

Para crear nuestro primer jugador, vamos a definir una clase que implemente la interfaz IStrategy, está clase va a ser muy simple, va a jugar siempre igual: apostando todo siempre, por ello la vamos a llamar *AggressiveStrategy*.

La implementación no puede ser más sencilla, la vemos en el ejemplo 10.1.

```
public class AggressiveStrategy implements IStrategy {

    private final String name;

    public AggressiveStrategy(String name) {
        this.name = name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public BetCommand getCommand(GameInfo<PlayerInfo> state) {
        return new BetCommand(TexasHoldEmUtil.BetCommandType.ALL_IN);
    }
}
```

Ejemplo 10.1

Ya tenemos nuestro primer jugador, ahora vamos a crear otro con un estilo de juego más impredecible, tan impredecible que va a jugar de forma aleatoria. Cómo no tenemos implementaciones por defecto para métodos que no usamos en la interfaz *IStrategy*, no hace falta implementar los métodos vacíos en esta clase.

Nuestra siguiente estrategia se va a llamar *RandomStrategy*.

En el ejemplo 10.2a y 10.2b tenemos el código de nuestro nuevo jugador.

```
public class RandomStrategy implements IStrategy {

    private static final Random RAND = new Random();
    private final String name;
    private double aggressivity = 0.5 + RAND.nextDouble() / 2;
    private BetCommand lastBet = null;

    public RandomStrategy(String name) {
        this.name = "Random-" + name;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void updateState(GameInfo<PlayerInfo> state) {
        lastBet = null;
    }

    @Override
    public String toString() {
        return String.join("{RandomStrategy-", name, "}");
    }

    @Override
    public void check(List<Card> communityCards) {
        lastBet = null;
    }

    private long getMaxBet(GameInfo<PlayerInfo> state) {
        if (aggressivity > 1.d) {
            return Long.MAX_VALUE;
        }
        long players = state.getPlayers().stream().filter(p -> p.isActive() ||
                         p.getState() == TexasHoldEmUtil.PlayerState.ALL_IN).count();
        double probability = 1.0D / players;
        long pot = state.getPlayers().stream().mapToLong(p -> p.getBet()).sum();
        return Math.round((probability * pot / (1 - probability)) * aggressivity);
    }
    ...
```

Ejemplo 10.2a

El método *getMaxBet()* trata de calcular las probabilidades que tiene el jugador de ganar un bote, aquí lo hace sin tener en cuenta sus propias cartas y el estado de la partida, pero la cuestión es tener un método de calcular hasta que cantidad se puede apostar sin tener en cuenta ningún otro factor.

```
    ...
    @Override
    public BetCommand getCommand(GameInfo<PlayerInfo> state) {
        PlayerInfo ownInfo = state.getPlayer(state.getPlayerTurn());
        calcAggressivity(state, ownInfo);
        long otherPlayerMaxBet = state.getPlayers().stream().max(
            (p0, p1) -> Long.compare(p0.getBet(), p1.getBet())).get().getBet();
        long minBet = Math.max(otherPlayerMaxBet – ownInfo.getBet(),
                               state.getSettings().getBigBlind());
        long maxBet = getMaxBet(state);
        long chips = ownInfo.getChips();
        BetCommand result;
        if (minBet > maxBet) {
            result = new BetCommand(BetCommandType.FOLD);
        } else if (maxBet >= chips) {
            result = new BetCommand(BetCommandType.ALL_IN);
        } else if (maxBet > minBet &&
                (lastBet == null || lastBet.getType() != BetCommandType.RAISE)) {
            result = new BetCommand(BetCommandType.RAISE, maxBet);
        } else if (otherPlayerMaxBet==state.getSettings().getBigBlind()||minBet==0) {
            result = new BetCommand(BetCommandType.CHECK);
        } else {
            result = new BetCommand(BetCommandType.CALL);
        }
        lastBet = result;
        return result;
    }
    private void calcAggressivity(GameInfo<PlayerInfo> state, PlayerInfo player) {
        long allChips = state.getPlayers().stream().filter(p -> p.isActive() ||
            p.getState() == TexasHoldEmUtil.PlayerState.ALL_IN).mapToLong(
            p -> p.getChips()).sum();
        long players = state.getPlayers().stream().filter(p -> p.isActive() ||
            p.getState() == TexasHoldEmUtil.PlayerState.ALL_IN &&
            p.getChips() > 0).count();
        long myChips = player.getChips();

        double proportion = (allChips - myChips) / players;
        aggressivity = (myChips / (proportion + myChips)) / 2 + 0.70d;
        if (myChips > (allChips - myChips)) {
            aggressivity = 1.1;
        }
    }
}
```

Ejemplo 10.2b

En función de la apuesta mínima que debe hacer, de la apuesta máxima y de la agresividad en cada momento se determina qué tipo de apuesta se realiza, aunque habría afinar un poco y calcular las probabilidades de victoria teniendo en cuenta nuestras cartas, las cartas comunitarias y los estilos de juego del resto de jugadores.

Un detalle, en el método toString() se utiliza String.join() otra novedad de Java 8.

Aquí damos por concluido este breve capítulo.

Capítulo 11: La interfaz gráfica

La última pieza que necesitamos para que el juego sea vistoso es la interfaz gráfica, la ubicaremos en la paquetería *org.poker.gui* y va a contener cuatro clases: *ImageManager, TextPrinter, TexasHoldEmTablePanel* y *TexasHoldEmView*.

La primera clase como su propio nombre indica la vamos a utilizar para obtener las imágenes que necesitaremos, como no van a ser muchas, las cargaremos una vez en memoria y las mantendremos ahí para utilizarlas bajo demanda, por este motivo no queremos tener varias instancias, así que vamos a incluir el patrón de diseño Singleton y cómo hay varias formas de implementarlo en Java, vamos a seguir las recomendaciones de Josua Bloch en el libro ***Effective Java*** y lo implementaremos mediante un enumerado.

En el ejemplo 11.1 tenemos el código.

```
@ThreadSafe
public enum ImageManager {
    INSTANCE;

    private static final Logger LOGGER = LoggerFactory.getLogger(ImageManager.class);

    public static final String IMAGES_PATH = "/images/";
    private final Map<String, Image> images = new HashMap<>();

    private ImageManager() {
    }

    public synchronized Image getImage(String imageFile) {
        Image image = images.get(imageFile);
        if (image == null) {
            try {
                image = ImageIO.read(getClass().getResource(imageFile));
                images.put(imageFile, image);
            } catch (IOException ex) {
                LOGGER.error("getImage \"" + imageFile + "\"", ex);
            }
        }
        return image;
    }
}
```

Ejemplo 11.1

Seguimos con la Clase *TextPrinter*, esta clase la vamos a emplear para pintar texto alineado con respecto a un punto, habrá dos tipos de alineado; vertical y horizontalmente, con tres posibilidades para cada tipo: arriba, medio y abajo para el alineado vertical. Con izquierda, centro y derecha para el alineado horizontal. Los tipos de alineamiento los modelaremos como Enumerados anidados, agregaremos los atributos para el tipo de fuente y para el color. Con sus Getters y Setters oportunos.

El código de la clase está en los ejemplos 11.2a y 11.2b.

```
@NoThreadSafe
public class TextPrinter {

    public enum VerticalAlign {
        TOP, MIDDLE, BOTTOM
    }

    public enum HorizontalAlign {
        LEFT, CENTER, RIGHT
    }

    private Font font;
    private Color color;
    private int width;
    private int height;
    private VerticalAlign vAlign = VerticalAlign.TOP;
    private HorizontalAlign hAlign = HorizontalAlign.LEFT;

    public Font getFont() {
        return font;
    }
    public void setFont(Font font) {
        this.font = font;
    }
    public int getWidth() {
        return width;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public VerticalAlign getVerticalAlign() {
        return vAlign;
    }
    public void setVerticalAlign(VerticalAlign vAlign) {
        this.vAlign = vAlign;
    }
    ...
```

Ejemplo 11.2a

```
    ...
    public HorizontalAlign getHorizontalAlign() {
        return hAlign;
    }
    public void setHorizontalAlign(HorizontalAlign hAlign) {
        this.hAlign = hAlign;
    }
    public Color getColor() {
        return color;
    }
    public void setColor(Color color) {
        this.color = color;
    }

    private int getOffSetX(int widthText){
        int result = 0;
        if (hAlign == HorizontalAlign.CENTER){
            result = (width - widthText)/2;
        } else if (hAlign == HorizontalAlign.RIGHT){
            result = width - widthText;
        }
        return result;
    }

    private int getOffSetY(int ascent, int descent){
        int result = ascent;
        if (vAlign == VerticalAlign.MIDDLE){
            result = (height + ascent - descent)/2;
        } else if (vAlign == VerticalAlign.BOTTOM){
            result = height - descent;
        }
        return result;
    }

    public void print(Graphics g, String text, int x, int y) {
        g.setColor(color);
        g.setFont(font);
        FontMetrics fm = g.getFontMetrics(font);
        int widthText = fm.stringWidth(text);
        g.drawString(text,
                     x + getOffSetX(widthText),
                     y + getOffSetY(fm.getAscent(), fm.getDescent()));
    }
}
```

Ejemplo 11.2b

El método *print()* es el que realiza la tarea de dibujar el texto, para ello se apoya en los cálculos de *getOffSetX()* y *getOffSetY()* para saber la posición final en la que se acabara de dibujar el texto, ya que el método drawString dibuja con alineamiento izquierda y verticalmente no se ajusta a ninguna de las opciones que queremos.

Ahora le toca el turno a la clase más extensa de este capítulo: *TexasHoldEmTablePanel*, el código lo tenemos repartido entre los ejemplos 11.3a-11.3h, en este primer fragmento vemos las constantes que necesitamos. Vamos a hacer que nuestro Panel implemente *IStragegy*, para interceptar a modo proxy toda comunicación con la estrategia del jugador.

```
public class TexasHoldEmTablePanel extends javax.swing.JPanel implements IStrategy{
    private static final int PLAYER_PADDING = 6;
    private static final Font DEFAULT_FONT = new Font(Font.SERIF, Font.BOLD, 12);
    private static final Font CHIPS_FONT = new Font(Font.SERIF, Font.PLAIN, 12);
    private static final Font PLAYER_STATE_FONT =new Font(Font.SERIF, Font.BOLD, 15);
    private static final Color DEFAULT_BORDER_PLAYER_COLOR = new Color(0xb06925);
    private static final Color TEXT_ROUND_COLOR = new Color(0x004000);
    private static final Color ACTIVE_PLAYER_FOREGROUND_COLOR = new Color(0x033E6b);
    private static final Color ACTIVE_PLAYER_BACKGROUND_COLOR = new Color(0x66a3d2);
    private static final Color PLAYER_TURN_FOREGROUND_COLOR = new Color(0x186b18);
    private static final Color PAYER_TURN_BACKGROUND_COLOR = new Color(0x50d050);
    private static final Color DEFAULT_PLAYER_BACKGROUND_COLOR = new Color(0xCD853F);

    private static final int DEFAULT_ROUND_CORNER_SIZE = 20;
    private static final String DOLLAR = " $";
    private static final int MAX_PLAYERS = 10;
    private static final int POTS_POSITION_INCREMENT = 12;
    private static final String CARDS_PATH = IMAGES_PATH + "cards/png/";

    private static final String CARDS_EXTENSIONS = ".png";
    private static final String CHIPS_PATH = IMAGES_PATH + "chips.png";
    private static final String DEALER_PATH = IMAGES_PATH + "dealer.png";
    private static final String BACKGROUND_PATH = IMAGES_PATH + "background.png";
    private static final String BACK_CARD = "back";
    private static final char[][] SUIT_SYMBOLS =
                              {{'♦', 'D'}, {'♠', 'S'}, {'♥', 'H'}, {'♣', 'C'}};
    private static final Point[] COMMUNITY_CARDS_POSITIONS = ...
    private static final Point[] PLAYER_POSITIONS = ...
    private static final Point[] PLAYER_BET_POSITIONS = ...

    private static final Point CHIPS_POSITION = new Point(570, 428);
    private static final Point CHIPS_TEXT_POSITION_INCREMENT = new Point(86, 4);
    private static final Dimension CARDS_DIMENSION = new Dimension(54, 75);
    private static final Dimension PLAYER_DIMENSION = new Dimension(100, 135);
    ...
```

Ejemplo 11.3a

Como vemos hay muchas constantes, nos sirven para parametrizar, todos los aspectos que he visto necesarios. Hay muchas líneas que son demasiado extensas y las he reducido con los tres puntos ya que de otro modo dificultaría excesivamente la legibilidad general.

En el proyecto hay recursos como las imágenes para las cartas, estas imágenes siguen una convención en los nombres, para identificar de forma simple la imagen de cada carta, de modo que podamos utilizar el método *toString()* del primer capítulo sustituyendo los símbolos de los palos por S, D, H y C.

A continuación vemos los atributos de la clase, entre ellos aparece uno que se llama *delegate* esta será la interfaz sobre la que vamos a delegar los métodos de la interfaz IStrategy, pero al actuar como proxy, vamos a extraer la información que nos permita mostrar el estado de la partida.

```
...
private final TextPrinter textPrinter = new TextPrinter();
private final List<Card> communityCards = new ArrayList<>(COMMUNITY_CARDS);
private List<Long> pots = new ArrayList<>();
private final PlayerInfo[] players = new PlayerInfo[MAX_PLAYERS];
private final BetCommandType[] bets = new BetCommandType[MAX_PLAYERS];
private final Map<String, Integer> playersByName = new HashMap<>();
private IStrategy delegate;
private long betRound = 0;
private long maxBet = 0;
private int playerTurn = -1;
private int dealer = 0;
private int round = 0;

public TexasHoldEmTablePanel() {
}

public void setStrategy(IStrategy delegate) {
    this.delegate = delegate;
}
private static String getCardPath(Card c) {
    String cardString = BACK_CARD;
    if (c != null) {
        cardString = c.toString();
        for (char[] suitSymbol : SUIT_SYMBOLS) {
            cardString = cardString.replace(suitSymbol[0], suitSymbol[1]);
        }
    }
    return CARDS_PATH.concat(cardString).concat(CARDS_EXTENSIONS);
}

private void paintCommunityCards(Graphics2D g2) {
    int i = 0;
    for (Card c : communityCards) {
        Point p = COMMUNITY_CARDS_POSITIONS[i++];
        String cardPath = getCardPath(c);
        g2.drawImage(ImageManager.INSTANCE.getImage(cardPath), p.x, p.y, null);
    }
    int roundX = COMMUNITY_CARDS_POSITIONS[
            (COMMUNITY_CARDS_POSITIONS.length)/2].x
            + CARDS_DIMENSION.width / 2;
    int roundY = COMMUNITY_CARDS_POSITIONS[0].y - 2 * PLAYER_PADDING;
    g2.setColor(TEXT_ROUND_COLOR);
    textPrinter.setFont(PLAYER_STATE_FONT);
    textPrinter.setVerticalAlign(TextPrinter.VerticalAlign.BOTTOM);
    textPrinter.setHorizontalAlign(TextPrinter.HorizontalAlign.CENTER);
    textPrinter.print(g2, "Ronda " + round, roundX, roundY);
}
...
```

Ejemplo 11.3b

Agregamos métodos de utilidades para pintar en capas, primero el fondo, luego las fichas, las cartas y finalmente los jugadores.

```
...
private void paintBackground(Graphics2D g2) {
    g2.drawImage(ImageManager.INSTANCE.getImage(BACKGROUND_PATH), 0, 0, null);
}

private void paintChips(Graphics2D g2) {
    Image chips = ImageManager.INSTANCE.getImage(CHIPS_PATH);
    g2.setColor(Color.BLACK);
    textPrinter.setFont(DEFAULT_FONT);
    textPrinter.setVerticalAlign(TextPrinter.VerticalAlign.TOP);
    textPrinter.setHorizontalAlign(TextPrinter.HorizontalAlign.RIGHT);
    int x = CHIPS_POSITION.x - (pots.size() * POTS_POSITION_INCREMENT) / 2;
    for (Long pot : pots) {
        g2.drawImage(chips, x, CHIPS_POSITION.y, null);
        textPrinter.print(g2,
            pot + DOLLAR,
            x + CHIPS_TEXT_POSITION_INCREMENT.x,
            CHIPS_POSITION.y + CHIPS_TEXT_POSITION_INCREMENT.y);
        x += POTS_POSITION_INCREMENT;
    }
    for (int i = 0; i < players.length; i++) {
        if (players[i] != null && players[i].getBet() > betRound) {
            Point p = PLAYER_BET_POSITIONS[i];
            g2.drawImage(chips, p.x, p.y, null);
            textPrinter.print(g2,
                (players[i].getBet() - betRound) + DOLLAR,
                p.x + CHIPS_TEXT_POSITION_INCREMENT.x,
                p.y + CHIPS_TEXT_POSITION_INCREMENT.y);
        }
    }
    g2.drawImage(ImageManager.INSTANCE.getImage(DEALER_PATH),
                 PLAYER_BET_POSITIONS[dealer].x,
                 PLAYER_BET_POSITIONS[dealer].y, null);
}

private void setCommunityCards(List<Card> cards) {
    communityCards.clear();
    communityCards.addAll(cards);
}

private void paintPlayers(Graphics2D g2) {
    g2.setStroke(new BasicStroke(2));
    for (int i = 0; i < players.length; i++) {
        paintPlayer(g2, i);
    }
}
...
```

Ejemplo 11.3c

Pintar los jugadores tiene una gran cantidad de posibilidades, si el sitio está vacío, si el jugador se ha quedado sin fichas, si está dentro o no de la ronda o si tiene el turno.

```
...
private void paintPlayer(Graphics2D g2, int i) {
    Point playerPosition = PLAYER_POSITIONS[i];
    Color borderPlayerColor = DEFAULT_BORDER_PLAYER_COLOR;
    if (players[i] != null) {
        Color backgroundPlayerColor = DEFAULT_PLAYER_BACKGROUND_COLOR;
        if (bets[i] != null || players[i].getChips() > 0) {
            if (i == playerTurn) {
                backgroundPlayerColor = PAYER_TURN_BACKGROUND_COLOR;
                borderPlayerColor = PLAYER_TURN_FOREGROUND_COLOR;
            } else {
                backgroundPlayerColor = ACTIVE_PLAYER_BACKGROUND_COLOR;
                borderPlayerColor = ACTIVE_PLAYER_FOREGROUND_COLOR;
            }
            g2.setColor(backgroundPlayerColor);
            g2.fillRoundRect(playerPosition.x, playerPosition.y, ...);
            if (players[i].isActive() ||
                players[i].getState() == TexasHoldEmUtil.PlayerState.ALL_IN) {
                int y = playerPosition.y + PLAYER_DIMENSION.height - ...
                g2.drawImage(ImageManager.INSTANCE.getImage(...);
                g2.drawImage(ImageManager.INSTANCE.getImage(...);
            }
            if (bets[i] != null) {
                String text = bets[i].name().replace("_", " ");
                g2.setColor(borderPlayerColor);
                textPrinter.setFont(PLAYER_STATE_FONT);
                textPrinter.setVerticalAlign(MIDDLE);
                textPrinter.setHorizontalAlign(CENTER);
                textPrinter.print(g2, text, ...
            }
        } else {
            g2.setColor(backgroundPlayerColor);
            g2.fillRoundRect(playerPosition.x, playerPosition.y, ...);
        }
        g2.setColor(Color.white);
        textPrinter.setFont(DEFAULT_FONT);
        textPrinter.setVerticalAlign(TextPrinter.VerticalAlign.TOP);
        textPrinter.setHorizontalAlign(TextPrinter.HorizontalAlign.CENTER);
        textPrinter.print(g2, players[i].getName(), ...);
        if (players[i].getChips() > 0) {
            textPrinter.setFont(CHIPS_FONT);
            g2.setColor(borderPlayerColor);
            textPrinter.setHorizontalAlign(TextPrinter.HorizontalAlign.RIGHT);
            textPrinter.setVerticalAlign(TextPrinter.VerticalAlign.BOTTOM);
            textPrinter.print(g2, players[i].getChips() + " $", ...);
        }
    }
    g2.setColor(borderPlayerColor);
    g2.drawRoundRect(playerPosition.x, playerPosition.y, ...);
}
...
```

Ejemplo 11.3d

El método *paintComponent()* es el que hay que sobrescribir heredado de *JPanel* para conseguir que se pinte la mesa de Póquer tal y como nosotros queremos.

Aquí comenzaremos a definir los métodos que hay que implementar para la interfaz *IStrategy*, en el método *check()* se llama a *calculatePots*() veremos más adelante como calcular ellos botes de un modo diferente al que se implementó en el capítulo 8, en la clase *ShowDownState*.

```
...
public synchronized void paintComponent(Graphics g) {
    Graphics2D graphics2d = (Graphics2D) g;
    graphics2d.setRenderingHint(KEY_ANTIALIASING, VALUE_ANTIALIAS_ON);
    paintBackground(graphics2d);
    paintCommunityCards(graphics2d);
    paintChips(graphics2d);
    paintPlayers(graphics2d);
}
public String getName() {
    return delegate.getName();
}

public BetCommand getCommand(GameInfo<PlayerInfo> state) {
    playerTurn = positionConverter(state, state.getPlayerTurn());
    updatePlayerInfo(state);
    repaint();
    return delegate.getCommand(state);
}
private void updatePlayerInfo(GameInfo<PlayerInfo> state) {
    maxBet = 0;
    for (PlayerInfo player : state.getPlayers()) {
        int pos = playersByName.get(player.getName());
        if (players[pos] != null) {
            players[pos].setBet(player.getBet());
            players[pos].setChips(player.getChips());
            players[pos].setCards(player.getCards());
            players[pos].setState(player.getState());
            players[pos].setErrors(player.getErrors());
            maxBet = Math.max(maxBet, players[pos].getBet());
        }
    }
}
public synchronized void check(List<Card> communityCards) {
    betRound = maxBet;
    setCommunityCards(communityCards);
    for (int i = 0; i < bets.length; i++) {
        if (bets[i] != null &&
            bets[i] != TexasHoldEmUtil.BetCommandType.ALL_IN) {
            bets[i] = null;
        }
    }
    pots = calculatePots(players);
    repaint();
    delegate.check(communityCards);
}
...
```

Ejemplo 11.3e

```
...
public synchronized void onPlayerCommand(String pn, BetCommand betCommand) {
    int pos = playersByName.get(pn);
    PlayerInfo player = players[pos];
    player.setBet(player.getBet() + betCommand.getChips());
    player.setChips(player.getChips() - betCommand.getChips());
    player.setState(TexasHoldEmUtil.convert(betCommand.getType()));
    maxBet = Math.max(maxBet, player.getBet());
    bets[pos] = betCommand.getType();
    delegate.onPlayerCommand(pn, betCommand);
    playerTurn = nextPlayerTurn(players, bets, maxBet, pos);
    repaint();
}

private static int nextPlayerTurn(PlayerInfo[] players, BetCommandType[] bets,
                                  long maxBet, int currentPlayerTurn) {
    int i = (currentPlayerTurn + 1) % players.length;
    while (i != currentPlayerTurn) {
        if (players[i] != null &&
                players[i].isActive() &&
                (bets[i] == null || players[i].getBet() < maxBet)) {
            long activePlayers = Arrays.stream(players)
                 .filter(p -> p != null &&
                  (p.isActive() || p.getState() == PlayerState.ALL_IN))
                 .count();
            if (activePlayers == 1){
                return -1;
            } else {
                return i;
            }
        }
        i = (i + 1) % players.length;
    }
    return -1;
}

private int positionConverter(GameInfo<PlayerInfo> state, int i) {
    int result = -1;
    if (i >= 0 && i < state.getNumPlayers()) {
        result = playersByName.get(state.getPlayer(i).getName());
    }
    return result;
}
...
```

Ejemplo 11.3f

En estos métodos se recibe la apuesta de un jugador (puede ser el mismo) y se actualiza el atributo que indica el jugador que tiene el turno.

El método *positionConverter()* lo que hace es obtener la posición de un jugador en la mesa ya que en el juego es posible que disminuya el número de jugadores y por lo tanto los índices relativos cambian, sin embargo queremos que mantengan la posición en la mesa con la que se empezó la partida.

Al actualizar el estado de la partida, hay determinadas situaciones que pueden ocurrir, como por ejemplo que sea la primera vez que se llama este método, con lo que aún no tenemos posicionados los jugadores en sus sitios, otra posibilidad es que muestren las cartas, que se acabe de saltar de ronda o que la partida haya terminado. La información de los jugadores habrá que actualizarla y tratar cada caso, a excepción del caso en el que la ronda comienza hay que extraerlo en otro método privado debido a la complejidad.

```
...
public synchronized void updateState(GameInfo<PlayerInfo> state) {
    if (state.getGameState() == GameState.PRE_FLOP) {
        updateStatePreFlop(state);
    }
    playerTurn = positionConverter(state, state.getPlayerTurn());
    updatePlayerInfo(state);
    if (round != state.getRound() || state.getGameState() == GameState.END) {
        if (state.getGameState() != GameState.END) {
            setCommunityCards(state.getCommunityCards());
            pots.clear();
        }
        Arrays.fill(bets, null);
    }
    round = state.getRound();
    repaint();
    delegate.updateState(state);
}

private void updateStatePreFlop(GameInfo<PlayerInfo> state) {
    betRound = 0;
    int i = 0;
    if (playersByName.isEmpty()) {
        for (PlayerInfo player : state.getPlayers()) {
            players[i] = new PlayerInfo();
            players[i].setName(player.getName());
            playersByName.put(player.getName(), i++);
        }
    } else {
        Set<String> currentPlayers = state.getPlayers().stream()
            .map(p -> p.getName()).collect(Collectors.toSet());
        for (i = 0; i < players.length; i++) {
            if (players[i]!=null && !currentPlayers.contains(players[i].getName()){
                players[i].setBet(0);
                players[i].setChips(0);
                players[i].setState(TexasHoldEmUtil.PlayerState.OUT);
            }
        }
    }
    pots = new ArrayList<>();
    Arrays.fill(bets, null);
    dealer = positionConverter(state, state.getDealer());
}
...
```

Ejemplo 11.3g

Aquí tenemos el método *calculatePots()*, ya hemos comentado que se puede calcular los botes con distintos algoritmos, pues aquí vemos otra propuesta, consisten en fraccionar las apuestas de cada jugador entre los distintos botes posibles, pero primero habrá que calcular cada bote, pues bien el número de botes es el número de jugadores que apostaron todo (sin repetir) y si hay jugadores que han apostado más que el mayor de los All-in o ningún jugador ha apostado todo, esas fichas pasaran un nuevo bote.

Después, fragmentamos las apuestas de los jugadores de forma que vayan cubriendo los botes con los valores de referencia, en este punto sería fácil saber que jugador opta a que bote, aunque en esta clase tal cual está planteada no resulta de interés.

Finalmente nos quedamos con los botes que tengan contengan fichas.

```
...
private static List<Long> calculatePots(PlayerInfo[] players) {
    List<PlayerInfo> playersList = Arrays.stream(players)
        .filter(p -> p != null)
        .collect(Collectors.toList());

    List<Long> sortedAllIn = playersList.stream()
        .filter(p -> p.getState() == PlayerState.ALL_IN)
        .map(p -> p.getBet())
        .sorted()
        .distinct()
        .collect(Collectors.toList());

    sortedAllIn.add(Long.MAX_VALUE);
    long[] quantities = new long[sortedAllIn.size()];
    playersList.stream().forEach(p -> {
        long bet = p.getBet();
        long quantity = bet;
        long last = 0;
        for (int i = 0; i < sortedAllIn.size() && quantity > 0; i++) {
            long diff = Math.min(sortedAllIn.get(i) - last, quantity);
            quantities[i] += diff;
            quantity -= diff;
            last = sortedAllIn.get(i);
        }
    });
    List<Long> pots = new ArrayList<>();
    IntStream.range(0, quantities.length)
        .filter(i -> quantities[i] > 0)
        .forEach(i -> pots.add(quantities[i]));
    return pots;
}
```

Ejemplo 11.3h

Como vemos siempre hay muchas formas de hacer la misma cosa, incluso utilizando las mismas herramientas y tecnologías. Nuestro objetivo debería ser quedarnos con la más opción más simple, y si supusiese algún problema, por rendimiento o consumo de recursos pues optar por la siguiente.

Después de crear la clase *TexasHoldEmTablePanel* que extiende de *JPanel*, vamos a construir una clase que extienda de *JFrame*, donde podamos ubicar la clase anterior, la disposición de los elementos va a ser muy sencilla, ya que el único panel que va a contener es el anterior, lo vamos a llamar *TexasHoldEmView* y va a tener una resolución de 1280x800.

En los ejemplos 11.4a y 11.4b tenemos el código de *TexasHoldEmView*,

```
public class TexasHoldEmView extends javax.swing.JFrame {

    private static final int WINDOW_HEIGHT = 800;
    private static final int WINDOW_WITH = 1280;
    private static final String WINDOW_TITLE = "J Poker";
    private static final String WINDOW_ICON = IMAGES_PATH + "poker-chip.png";

    private TexasHoldEmTablePanel jTablePanel;

    public TexasHoldEmView(IStrategy delegate) {
        initComponents();
        setTitle(WINDOW_TITLE);
        setIconImage(ImageManager.INSTANCE.getImage(WINDOW_ICON));
        setBounds(0, 0, WINDOW_WITH, WINDOW_HEIGHT);
        jTablePanel.setStrategy(delegate);
    }

    public IStrategy getStrategy() {
        return jTablePanel;
    }
    ...
```

Ejemplo 11.4a

Vemos como en esta clase se recibe la estrategia en la que finalmente vamos a delegar las acciones a tomar.

Si quisiésemos adaptar la interfaz gráfica para sustituir el jugador automático por el usuario, este sería un buen lugar donde hacerlo, habría que agregar botones y gestionar memoria compartida para sincronizar la invocación del método *getCommand()* de *IStrategy()* con las acciones realizadas por el usuario.

El método *initComponents()* lo genera y edita Netbeans mientas utilizamos el asistente para las interfaces gráficas, normalmente se encuentra colapsado y únicamente se ve una línea.

```
...
private void initComponents() {
    jTablePanel = new TexasHoldEmTablePanel();
    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
    jTablePanel.setPreferredSize(new Dimension(WINDOW_WITH, WINDOW_HEIGHT));
    javax.swing.GroupLayout jTablePanelLayout = new GroupLayout(jTablePanel);
    jTablePanel.setLayout(jTablePanelLayout);
    jTablePanelLayout.setHorizontalGroup(
        jTablePanelLayout.createParallelGroup(LEADING)
            .addGap(0, WINDOW_WITH, Short.MAX_VALUE)
    );
    jTablePanelLayout.setVerticalGroup(
        jTablePanelLayout.createParallelGroup(LEADING)
            .addGap(0, WINDOW_HEIGHT, Short.MAX_VALUE)
    );

    javax.swing.GroupLayout layout = new GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setHorizontalGroup(
        layout.createParallelGroup(LEADING)
            .addComponent(jTablePanel, javax.swing.GroupLayout.DEFAULT_SIZE,
                          WINDOW_WITH, Short.MAX_VALUE)
    );
    layout.setVerticalGroup(
        layout.createParallelGroup(LEADING)
            .addComponent(jTablePanel, javax.swing.GroupLayout.DEFAULT_SIZE,
                          WINDOW_HEIGHT, Short.MAX_VALUE)
    );
    pack();
}
```

Ejemplo 11.4b

Aquí damos por finalizado este capítulo, como hemos visto la interfaz gráfica se ha implementado de forma muy sencilla y no se ha seguido el siempre recomendable patrón de diseño MVC, no obstante se puede refactorizar fácilmente para conseguir llevar este diseño a un MVC en toda regla.

La pregunta es: ¿Sí ya está hecho y cumple su finalidad para que cambiarlo? Pues sencillo, para utilizar el motor del juego con otro tipo de interfaces, por ejemplo una web, utilizando Html5 como interfaz gráfica y comunicando los eventos del juego con el navegador mediante WebSockets.

La siguiente pregunta es: ¿Y por qué no lo has hecho? Pues porque el tiempo es un recurso escaso y únicamente pretendía tener un prototipo funcional con una interfaz gráfica muy simple.

Capítulo 12: Ensamblando componentes

Si unimos todas las piezas para echar el juego a andar necesitaremos crear una clase principal con el método *main()*, esta clase la llamaremos *MainController.*

Lo primero que hacemos es instanciar la clase *RandomStrategy* en la variable *strategyMain* para separarla del resto, agregamos las demás estrategias y mezclamos para que no estén siempre en la misma posición, ya que el estilo de juego de los jugadores a ambos lados puede condicionar bastante.

Preparamos la configuración de la partida, indicando los valores necesarios para poder ejecutar el juego y lanzamos su ejecución.

```
public final class MainController {

    private static final int PLAYERS = 10;

    public static void main(String[] args)throws InterruptedException,GameException {
        IStrategy strategyMain = new RandomStrategy("RandomStrategy-0");

        List<IStrategy> strategies = new ArrayList<>();
        strategies.add(strategyMain);
        for (int i = 1; i < PLAYERS; i++) {
            strategies.add(new RandomStrategy("RandomStrategy-"+String.valueOf(i)));
        }
        Collections.shuffle(strategies);
        Settings settings = new Settings();
        settings.setMaxErrors(3);
        settings.setMaxPlayers(PLAYERS);
        settings.setMaxRounds(1000);
        settings.setTime(500);
        settings.setPlayerChip(5000L);
        settings.setRounds4IncrementBlind(20);
        settings.setSmallBlind(settings.getPlayerChip() / 100);
        IGameController controller = new GameController();
        controller.setSettings(settings);
        for (IStrategy strategy : strategies) {
            controller.addStrategy(strategy);
        }
        controller.start();
    }
}
```

Ejemplo 12.1

Si queremos lanzar el juego con la interfaz gráfica que hemos creado en el capítulo anterior, bastaría con agregar las tres líneas sombreadas en amarillo del ejemplo 12.2 como tenemos separada del resto la estrategia *strategyMain*, utilizamos esta como base, será la clase delegada que utilizará el Panel de la interfaz gráfica. Por lo tanto veremos que cartas recibe y como decide jugarlas.

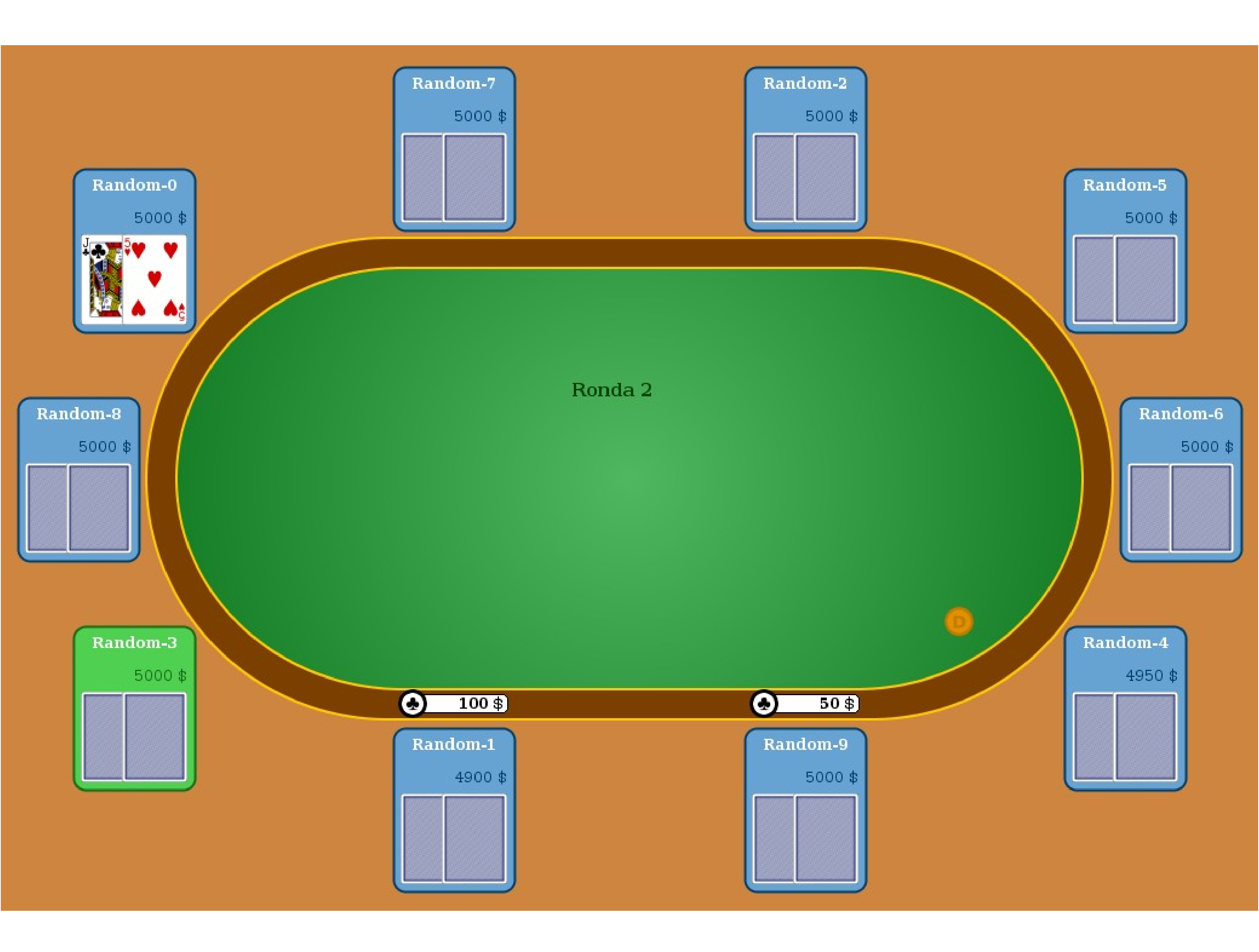
```
public final class MainController {

    private static final int PLAYERS = 10;

    public static void main(String[] args)throws InterruptedException,GameException {
        IStrategy strategyMain = new RandomStrategy("RandomStrategy-0");
        TexasHoldEmView texasHoldEmView = new TexasHoldEmView(strategyMain);
        texasHoldEmView.setVisible(true);
        strategyMain = texasHoldEmView.getStrategy();
        List<IStrategy> strategies = new ArrayList<>();
        strategies.add(strategyMain);
        for (int i = 1; i < PLAYERS; i++) {
            strategies.add(new RandomStrategy("RandomStrategy-"+String.valueOf(i)));
        }
        Collections.shuffle(strategies);
        Settings settings = new Settings();
        settings.setMaxErrors(3);
        settings.setMaxPlayers(PLAYERS);
        settings.setMaxRounds(1000);
        settings.setTime(500);
        settings.setPlayerChip(5000L);
        settings.setRounds4IncrementBlind(20);
        settings.setSmallBlind(settings.getPlayerChip() / 100);
        IGameController controller = new GameController();
        controller.setSettings(settings);
        for (IStrategy strategy : strategies) {
            controller.addStrategy(strategy);
        }
        controller.start();
    }
}
```

Ejemplo 12.2

Este es el resultado de la ejecución con todos los elementos integrados con la interfaz gráfica:



Con esto damos por concluido el juego, se han quedado numerosas novedades de Java 8 en el tintero como la nueva API de java.time, motor JavaScript, nuevas excepciones o métodos en tipos String, Integer, etc... Lo importante es tomar contacto con Java 8 y experimentar. En github está el código, en el repositorio: https://github.com/dperezcabrera/jpoker.git

Bibliografía

Effective Java, segunda edición (2008) de Joshua Bloch.

Clean Code: A Handbook of Agile Software, primera edición (2008) de Robert C. Martin

Documentación oficial de slf4j: http://www.slf4j.org/docs.html

Documentación oficial de logback: http://logback.qos.ch/documentation.html

Tutorial de Colecciones http://docs.oracle.com/javase/tutorial/collections/

Using Assertions in Java Technology http://www.oracle.com/us/technologies/java/assertions-139853.html

Java Generics and Collections, primera edición (2006) de Maurice Naftalin y Philip Wadler,

Next Generation Java Testing: TestNG and Advanced Concepts (2007) de Cédric Beust y Hani Suleiman.

JUnit in Action, Segunda edición (2010)de Petar Tahchiev, Felipe Leme, Vincent Massol y Gary Gregory.

Práctical Unit Testing with JUnit and Mockito, primera edición (2013) de Tomek Kaczanowski.

Effective Unit Testing: A guide for Java developers, primera edición (2013) de Lasse Koskela.

Test Driven: TDD and Acceptance TDD for Java Developers, primera edición (2007) de Lasse Koskela.

Test Driven Development. **By Example**, primera edición (2008) de Kent Beck.

Domain Specific Languages, primera edición (2010) de Martin Fowler.

The Cucumber Book: Behaviour-Driven Development for Testers and Developers (Pragmatic Programmers), primera edición (2012) de Matt Wynne y Aslak Hellesoy.

Cucumber Recipes: Automate Anything with BDD Tools and Techniques, primera edición (2013) de Ian Dees, Matt Wynne y Aslak Hellesoy.

Refactoring: Improving the Design of Existing Code, primera edición (2008) de Martin Fowler, Kent Beck, John Brant y William Opdyke.

Java Theory and Practice: Decorating with Dynamic Proxies de Brian Goez www.ibm.com/developerworks/library/j-jtp08305/

AspectJ in Action: Enterprise AOP with Spring Applications primera edición (2009) de Ramnivas Laddad.

Design Patterns: Elements of Reusable Object-Oriented Software, primera edición (1994) de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides.

Inversion of Control Containers and the Dependency Injection pattern de Martin Fowler http://martinfowler.com/articles/injection.html

Java concurrency in practice, primera edición (2006) de Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea y David Holmes.

The Java Language Specification, Java SE 8 Edition (Java Series), primera edición (2014) de James Gosling y Bill Joy.

Java 8 in Action: **Lambdas, Streams, and functional-style programming**, primera edición (2014) de Raoul-Gabriel Urma, Mario Fusco y Alan Mycroft.

Java 8. Los Fundamentos Del Lenguaje Java, primera edición (2014) de Thierry Groussard.

JavaTM Puzzlers: Traps, Pitfalls, and Corner Cases, primera edición (2005) de Josua Bloch y Neal Gafter.

Puzzles in Java : Shaping Beginners, primera edición (2013) de Kathiravan Udayakumar.

Programming in Scala: A Comprehensive Step-by-Step Guide, primera edición (2011) de Martin Odersky, Lex Spoon y Bill Venners.

Groovy in Action, primera edición (2007) de Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge y Jon Skeet.

The Definitive Guide to Java Swing, primera edición (2011) de John Zukowski.